HYBRID MEMORIES FOR ENERGY EFFICIENT COMPUTING SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Wing-kei Shanahan Yu

February 2016

HYBRID MEMORIES FOR ENERGY EFFICIENT COMPUTING SYSTEMS

Wing-kei Shanahan Yu, Ph.D.

Cornell University 2016

To sustain processor performance, demand for memory capacity and bandwidth continues to grow. Computer architects use different memories, structured in a hierarchy, to build memory systems to satisy that demand. Architects design hierarchies to take advantage of trade-offs inherent to the different memory technologies available.

New memory technologies, such as non-volatile memories, bring different trade-offs into the design space. Interestingly, non-volatile memory technologies have a set of trade-offs that complement existing memory technology well. As a result, architects have proposed *hybrid memory* designs, using both traditional and new memories in the same memory structure, to exploit memory technology advantages and mitigate their disadvantages. Using disparate memories in tandem, at the same level of hierarchy, can unlock efficiencies and new capabilities unseen with monolithic memory arrays.

In this thesis I describe and categorize a landscape of hybrid memory organizations, and then describe 3 concrete projects we undertook to demonstrate the advantages of certain hybrid organizations. I discuss a hybrid SRAM-Flash memory that reduces energy usage and brings non-volatile save and restore operations within reach of energy-limited platforms, a hybrid SRAM-DRAM memory that can replace SRAM-only register files with the same capacity and performance but at reduced area and energy costs, and a hybrid SRAM-MRAM cache using compression to reduce overall cache energy.

BIOGRAPHICAL SKETCH

Wing-kei (KK) Yu is a native of Houston, Texas, and is a thoroughly indoctrinated Texan. He became interested in computer engineering during primary school, after his friend Omer asked him how fast his family's new 486 PC was (50 MHz). During his undergraduate education at Rice University, he had a brief flirtation with becoming a physics major, but remained in electrical engineering, whetting his appetite by studying semiconductor device physics in addition to computer engineering. After graduating, he studied abroad for a year in Hong Kong, where he learned to appreciate foreign viewpoints of the USA, public transportation, and photography.

Entering Cornell University, he aimed to use his training in devices and computer architecture to find new methods and solutions to computing problems. While there he met many incredibly helpful and intelligent people. Aside from being trained to carry out research, he is grateful to have been part of a vibrant faith community that helped to nourish his relationship with the Christian God and the Church. Unexpectedly, he also discovered a new passion for the world of business and how some of its lessons can be applied to the ivory tower, a discovery he intends to share.

Some things KK enjoys in his free time are family, friends, chatting, a good drink, and planespotting.

This thesis is dedicated to my parents, who waited a long time.

ACKNOWLEDGEMENTS

An incredible amount of people have helped me to finish this thesis and have supported and continue to support me through this phase of life. If I could list every single one of you, my thesis would be twice as long. I am grateful for everyone who touched my life.

I thank the triune God: Father, Son, and Holy Spirit for everything. I grew as a Christian here, which would be a surprise to my younger self. As scientists we search for the answers to the "ultimate questions," which we attempt to answer with science, as our training has taught us to do. But there are fields where we cannot apply science to deduce the truth. History is the most visible, as we cannot repeat or run controls against historical events, but we take them as truth for other reasons, and use that information to live our lives and to make decisions. Exploring and discovering faith and the afterlife, while realizing I had to use other tools besides science to accept or reject their claims, was a literal life-changing experience. I would be happy to discuss this with anyone. I am grateful that God has brought me here to Cornell, to grow and to serve those I am lucky enough to know.

I certainly owe a great deal to my parents, who have always supported me, even as the years kept going by. I am thankful they continued to encourage me to finish this degree, and for sticking with me. Similarly, I am lucky to have my brother and sister, who are always around, and are never afraid to tell it to me straight. And, from the start of my time at Cornell, a wonderful woman was placed into my life, Xiaoyue Chen. I owe her much, because she believed in me, and continues to.

Of course, it was an honor to work and interact with my PhD advising committee:

G. Edward Suh. My primary advisor, who showed me what good research is, an appreciation for computer security, and was very gracious to me.

Edwin Kan, who led me through the world of fabrication and devices, and with whom I worked with on almost all my projects.

Rajit Manohar, who showed me the wonders of asynchronous computing, building giant chips, and for several enjoyable chats on topics academic and otherwise.

I am in debt to my committee for all of their help and support. They have taught and showed me so much about the profession of professorship, and how to drive oneself, drive a research group, and drive a life outside of that too. I am grateful I studied under them, and wish them and the future students they mentor all the best.

I was fortunate to be a part of the CSL (Computer Systems Laboratory), an umbrella research group where the people and environment came together to foster many undergraduate and graduate students' careers, as well as being a permanent home for the faculty who shepherded the group. I want to thank all who have passed through CSL for shaping the group and being a part of my experience.

To the entering graduate class of 2007: it was a great ride! We are finally all done! Thanks to all of you – Ben Hill, Derek Lockhart, Janani Mukundan, Raymond Huang, and Saugata Ghose – for entering, starting off, and going through CSL together.

I am grateful for all of my officemates in Upson 365 UC. Rob Karmazin was the most permanent resident, and it was a true constant to be able to turn around

and find Rob there, staring at layout, code, and/or a Prince Polo bar. I am so grateful to know Benjamin Hill, as he shared a passion for education and is the truest gentleman (in every sense of the word) I have yet to know. Derek Lockhart taught me that Google isn't all evil, and I hope I softened his heart towards Microsoft too. Dan Deng showed me a lot about graduate life. I remember one time he declared that a graduate student should always be able to give a presentation. It's true. Jon Tse kept me aware and prepared for the end times, and is a good friend. Later on, I was able to see Yao Wang freshly arrive in the States and become an inquisitive and thoughtful researcher and companion. I was glad to be a part of his welcome to America. And too briefly I shared the office with Kyle Wecker. Sorry for always typing too loudly! His daughter Alice and I share a birthday, exactly a quarter century apart. His love for her showed me something worth aiming for in life.

Outside of my office, I could always find Saugata Ghose being the bedrock of Upstate New York knowledge, when he wasn't always helping others. Thanks for being positive, always. Ji Kim showed me that engineers are weird and a bit out of touch with their emotions, but that I didn't have to be that way. Thanks for your patience and frankness. Shreesha Srinath, I appreciate you always being up for coffee and a chat. That goes for Steven Longfield too, though he is more of a tea drinker. Julia Karl provided a sounding board for graduate school improvement ideas, and brought a much-needed dose of the working world to me.

Across the hall I could find the rest of my research group, the Suh Research Group. I've spent much time there shooting the breeze, talking ideas, getting edits, giving edits, informally mentoring, and having some fun too. Raymond Huang was always up for dinner and showed me the merits of doing things the

Canadian way. Andrew Ferraiuolo, master computer hacker, entertained many of my ridiculous ideas. Dan Lo entertained even more of those crazy ideas, showed me lots of cool hobbies, that you could be an outdoorsman and a grad student at the same time, and put up with my graduate school idealism. We also designed some book covers once. The rest of the group I did not know as well, yet we shared meals and meetings, and taught and learned from each other nevertheless. Mohamed Ismail, Tao Chen, and Taejoon Song, I wish you all the best.

I thank my PhD co-authors for great times and for working with me. Shantanu Rajwade designed and simulated the SRAM-Flash circuits. Sarah Xu designed and fabricated our SRAM-DRAM and proved that Yinglei and I's Flash variatinos were actually random. Raymond Huang hacked GP-GPUSim in record time for the SRAM-DRAM project. Yinglei Wang and I worked together on Flash security, using actual hardware too. Jon Tse and Carlos Tadeo Ortega Otero had me join a project on memory for asynchronous chips. I'm thankful for my cross-country collaborators at UCSD, Laura Caulfield and Steve Swanson, for our Flash characterization work.

I owe a lot to the Masters students and undergraduates I worked with. Sung-En Wang and Bob Lian spent many nights designing SRAMs for me in Phillips 314. I mismanaged the considerable talents of Yu-Yuan Kuo, but he was gracious to me and designed and simulated lots of analog magic. Greg Malysa built a custom Flash board which would go on to spawn many many papers and is still being used, years after he left. Shuo Wu did statistical analysis for Yinglei and I. Scott Warren implemented our random number generator on an MSP430, and Max Spector tried to do so on an Android phone. I also enjoyed getting to know Yunchi Luo, Cathy Chen, Alexis Colton, and Jess Loeb. Thank

you all.

Outside of CSL, I spent quite a few days in the office of Albert Wang, Caroline Andrews, and Ben Johnson, and we would talk about radios and speakers, and sometimes digital logic. Chang-Hyuk Lee would come by too, and later on Suren Jayasuriya, ECE social coordinator extraordinaire, took up residence there. Thanks for letting me visit. When I wasn't bothering that office of Al Molnar's students, I was chatting with the other one. Hazal Yksel and Rose Agger kept me entertained then. Ying Niu was not in Al's group, but I found a friend in her too. Xiao Wang, an ECE legend, could always be counted on as well.

It was my pleasure to know faculty and staff in the ECE department as well. Scott Coldren, Patty Gonyea, and T. Daniel Richter smoothed the way many times over. The energy and passion of Chris Batten and Al Molnar kept me excited and engaged. Christoph Studer's excitement wore off on me too. Cliff Pollock is a fellow Rice Owl and has great stories. Alyssa Apsel backed me when times got tough, and we found a common goal in trying to professionalize the graduate student working mindset.

I am grateful to the tax-paying public, the National Science Foundation, and other US government sources, because they provided financially for me to study and live here.

When I was not working, I found a home in the Christian community at Cornell. I am grateful for the Graduate Christian Fellowship, my first Christian home at Cornell. My first friends there – Tim Condon, Alvina Lin Condon, Neela Babu, Nate Hansen, Katrina Lyon, Jennifer Lee, Hansen Hsu – I remember fondly over board games and Bible studies. Mike and Sarah Norman opened their home to me during Bible study as well.

I am overjoyed to have found a church home in First Ithaca Chinese Chris-

tian Church (FICCC) so quickly, right from the start. It has been my only church home and it has molded and seen me grow from just a face in the crowd to sticking around long past service, giving rides and being a part of the community. I was finally baptized here as well, in April 2013. I thank God for my pastors, Tony Hsu and Paul Epp. I enjoyed Pastor Tony's forceful sermons though I was not very active then. Pastor Paul has been nothing but a blessing to me. He is a pastor, mentor, friend, and prayer warrior. He and his wife Hildy and their children have opened their home to me many times as well. Thanks. More recently, Intern Pastor David has been someone I could easily relate to.

FICCC did not have many graduate students on the English side, but the ones that were there had a huge impact on me. Jenny Lee taught me how to be a busy graduate student and a busy church member and leader. Luke McDermott and I led Bible studies together, and tried to stay sane while juggling faith and work together. Christine Ouyang always made me feel welcome, and Songwei Chen taught me a lot about patience. On the Chinese side I made many friends, too numerous to list! Thanks for putting up with my broken Mandarin, and welcoming me.

I am immensely grateful to have been a part of an undergraduate fellowship, Chinese Bible Study (CBS). It provided a community and a place where I could be a mentor in school, and to learn from strong Christians, though younger than I. I cannot even begin to list those who have let me sit in their rooms, eat their food, worship with and serve with, and who have all kept me sane through my graduate career here. You have given me a second set of college friends.

Late in my graduate career, I became interested in the business world and how its lessons could improve academia. I owe a lot of that to conversations with business-minded Cornell alums: Iris Wen, Tiffany Yu, and Rachel Pang. I

have also talked to many graduate students about this, and am thankful for their thoughts. I consider this one of my most valuable take-aways from my time here. I want to acknowledge Colleen McLinn, who heads an initiative at Cornell to improve graduate school education, mentorship, and operation. I look forward to working on a seminar on improving graduate student entrepreneurship and mindset that her initiative inspired me to create.

While finishing up, I went to New York City quite often, to de-stress and create some space for myself. I was able to do that so informally because of the generosity of my Aunt Elena, and cousins Justin and Jinn, who let me stay so often. And when in New York, I found a constant friend in Rachel Pang, who showed me around the city, and who reminded me of my fondness for airliners.

Lastly, in the world of the internet, I found companionship and good conversation online too. During the last stretch I wanted to thank a few of my conversation partners for being available almost anytime: Franklin Leung, my oldest friend, who also finished a graduate degree, Jacqueline Chien, who started one, and Jamie Tsai, who thinks a lot.

The pleasure is all mine. Thank you all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The memory system as we know it has remained largely unchanged in basic form and structure since the advent of the hard drive. As the only non-volatile memory in most computing devices, the hard drive is the primary store of information. The structure of the memory hierarchy evolved to balance out the ever increasing disparity in performance between the hard drive and processor, with the addition of more levels of hierarchy, and by increasing the memory capacity of all levels of the hierarchy.

Even as memory capacity increases, processor demand on memory bandwidth and latency has never been satiated and continues to rise. Even the slowing of frequency and single-threaded IPC has not dented this trend – multi-core processors have only increased pressure on the memory systems of today, adding more cores that share a single memory interface.

More memory means more power and more energy. In fact, memory size is limited by power and energy today, especially in server machines. To increase memory capacity while retaining reasonable energy costs, system architects have constructed very clever organizations and solutions, ranging from improved caching and snooping algorithms to more efficiently use existing memory, to rearchitecting systems (i.e. introduction of NUMA and NUCA into processor and blade server design). However, the technological limitations of traditional DRAM and SRAM memory have remained a basic hurdle. In particular, static energy has become a larger percentage of energy usage, and DRAM and SRAM consume great amounts of static power.

	SRAM	DRAM	PCM	MRAM	Flash
Latency					
– Read	Fastest	Faster	Fast	Faster	Slow
– Write	Fastest	Fast	Slow	Slow	Slower
Energy					
– Static	Highest	High	Low	Low	Low
– Dynamic					
—- Read	Low	Low	Low	Low	Low
—- Write	Low	Low	High	High	High
Endurance	–	–	10^7	10^{15}	10^3
Area	High	Low	Low	Low	Very Low
Non-volatile	No	No	Yes	Yes	Yes

Table 1.1: Comparison of traditional and non-volatile memory characteristics.

Recently, the rise of rise of new memories that rival established DRAM and SRAM in terms of performance, yet consume much less energy, have enabled architects to try new techniques to bring more capacity to computing systems. These new memories are non-volatile, avoiding static energy and thereby achieving substantial energy savings compared to traditional volatile devices. This allows, for the same energy budget, more capacity to be installed, helping to satisfy the demand for more memory. The drawbacks of these new memories are that they are not quite as fast as DRAM or SRAM, are sensitive to the number of writes they can tolerate, and have steep write energy costs.

Table 1.1 shows a general comparison of traditional and non-volatile memory characteristics. Note that where SRAM and DRAM show weaknesses, the non-volatile memories – PCM, MRAM, and Flash – are often strong. For example, SRAM and DRAM have high static energy, while non-volatile memories have low static energy. These complementary characteristics indicate the possibility of using different types of memory together, exploiting their strengths

and minimizing their weaknesses.

In summary, three factors have converged to make hybrid memories an interesting design choice.

- Increasing pressure on memory subsystems as a result of Moore's law
- Increasing memory energy cost due to failure of Dennard scaling
- Rise of non-volatile memories with complementary energy characteristics

Architects have flocked to take advantage of capacity/energy trade-off these new memories offer. To mitigate the performance and write endurance disadvantages, they have tended to pair these new memories with traditional ones, often in a cache-like organization, with the new non-volatile memory as a large backing store and a small traditional memory as a cache. Using the non-volatile memory to replace large amounts of traditional memory saves energy overall, while using traditional memory mitigates the write energy and endurance disadvantages of non-volatile memory. This allows the construction of a memory array with a much larger capacity than an array composed of only traditional memory at a similar energy cost, while maintaining performance.

Note that increased capacity is not the only benefit that these new memories offer. Combining two different types of memory can allow the use of new functionalities, for example, the addition of non-volatile memory to a memory can enable fast and low-energy checkpointing.

Hybrid memories have received considerable interest from architects. The term "hybrid memory," at its broadest, simply means a memory structure composed of different memory types; using this sweeping classification allows us to encompass any system with more than one type of memory, including current

systems (which are typically SRAM-DRAM-NVM (hard drive, Flash) hybrids). This is not the meaning currently used in the computer architecture research community. Instead, the computer architecture community takes the term "hybrid memory" to mean that the memory at one specific level in the memory hierarchy is composed of two or more different types of memory. For example, an L2 cache formerly composed of only SRAM, and now composed of both SRAM and MRAM would be an example of hybrid memory. A hard drive with a small Flash cache (hybrid hard drives, as they are called) is a hybrid memory.

Interest in leveraging the various benefits of hybrid memories has driven researchers to propose myriad ideas and evaluate them, and has resulted in many publications in recent computer architecture venues. Over the past 6 years, numerous publications in the top four computer architecture conferences have involved a hybrid memory. Amidst the search for a universal memory, fears of DRAM scaling coming to an end, and recent developments in the non-volatile memory world (such as memristors), the amount of interest in hybrid memory and their possible benefits is only going to increase.

My thesis contributes the following to the hybrid memory research space:

- A categorization system for hybrid memories, in order to provide a vocabulary for the configurations of the hybrid memories being proposed in literature.
- Three hybrid memory projects, demonstrating energy efficiencies of using hybrid memories in architecture design:

  Hybrid SRAM-Flash for non-volatile computing

  This project uses a hybrid memory design using SRAM and Flash,

which are integrated at a very close level. As a result, data movement is extremely cheap compared to traditional solutions. The drastically reduced cost enables extremely quick and efficient non-volatile store and restore operations, which brings this capability within reach of power-limited domains.

Hybrid SRAM-DRAM for multi-threaded register files

Using a hybrid memory design comprised of SRAM and DRAM, this project replaces a very large SRAM register file with the smaller hybrid memory design, while keeping the same capacity. Taking into consideration the characteristics of fine-grained multi-threaded processing, we design an architecture that maintains performance while reducing register file energy.

Hybrid SRAM-MRAM for energy-efficient caches using compression

This hybrid memory design attacks weaknesses of SRAM-only and MRAM-only caches. Static energy use is a large liability for SRAM, while MRAM spends the majority of its energy in writes. Using a smaller amount of each type of memory in tandem to build a cache of the same overall capacity, we reduce energy usage by using a smaller SRAM to cut static energy as well as by using compression to convert expensive MRAM writes to SRAM writes.

The following chapters describe my contributions, in the order given. After each contribution is described, the thesis is concluded.

CHAPTER 2

CLASSIFICATION OF HYBRID MEMORIES

This chapter classifies and summarizes recent research in hybrid memory structures and new non-volatile memory (NVM) proposals, specifically in the computer architecture community, but not necessarily limited to that area. Researchers use various architectural design points and tradeoffs in order to exploit the energy and/or performance benefits of these new memory technologies. In proposing these hybrid memory designs, researchers have used many different ways to architect the memories used. To organize and better understand how these proposals relate to each other, we aim to place them in a broad design space, using a classification system.

In surveying the literature, we note that there exist three different design decisions that an architect needs to make when designing a hybrid memory. The three decisions can be structured as the following questions:

Internal Organization How do data flow into the separate memory structures inside the hybrid memory designs?

External View How are the internal memory structures inside the hybrid memory design shown to the outside world?

Level of Integration How are the memories connected together at the physical level?

We discuss each design decision in its own section below.

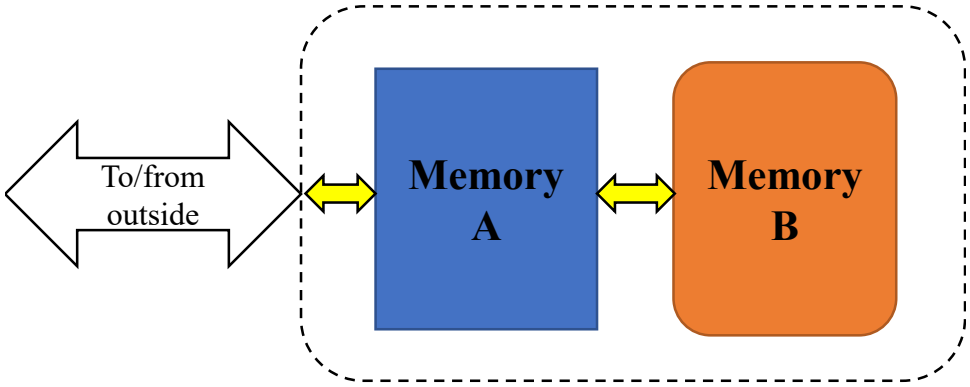


Figure 2.1: A serially organized hybrid memory design.

2.1 Internal Organization

One design decision that must be made is how to structure data movement between the individual memory structures inside the hybrid memory design.

A hybrid memory may be organized in a serial or parallel manner:

2.1.1 Serial

This is an internal organization in which the internal memories are linked in a cache-like manner: all data flow from one (usually larger, slower) memory array to another (usually smaller, faster) memory array before they are available to be accessed from the outside. Only one of the internal memories is in contact with the outside world.

This organization is shown in Figure 2.1.

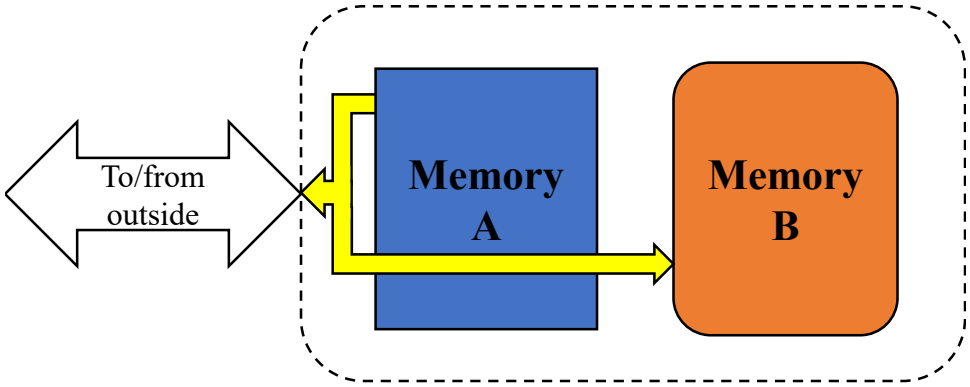


Figure 2.2: A parallel organized hybrid memory design.

2.1.2 Parallel

This is an internal organization where each memory is directly accessible from the outside. Data can flow directly from the array it is originally stored in without passing through a cache or other memory array. This organization is shown in Figure 2.2.

Note that the internal organization of a hybrid memory is orthogonal to the level of integration (i.e. per-cell or per-array integrated hybrid memories can be organized in a serial or in a parallel fashion).

2.2 External View

The external view describes how hardware and software not part of the hybrid memory itself is aware of the nature of the hybrid memory. In essence, this is how the hybrid memory is seen from the outside. There are two broad cate-

gories, exposed and hidden.

2.2.1 Hidden

Many hybrid or new non-volatile memory structures tend to be structured as drop-in replacements for existing levels of the hierarchy (i.e. on-chip SRAM caches are replaced, off-chip DRAM is replaced), partly for ease of integration into existing system architectures. In these cases, the hybrid nature of the memory is hidden from the outside system.

2.2.2 Exposed

Other proposed memory structures instead expose both memory types to the outside system, making both types independently accessible. Usually, exposing both memory types to an external view requires a parallel internal organization for ease of access.

2.3 Level of Integration

The level of integration criteria describes how the memories in a hybrid structure are organized at the physical level.

Most memories are built in monolithic arrays, and the technology process used to construct them is optimized for a certain array layout and memory cell. It logically follows that a hybrid memory might be composed of separate ar-

rays of different memories. This level of integration (separate arrays of different memories on the same physical substrate) we term *per-array* integration. Per-array integrated hybrid memory designs use multiple arrays of disparate memories on the same die. It is a common choice for integration because it leverages existing fabrication techniques, while still allowing relatively cheap data movement between memory types.

Some work proposes integrating disparate memories at a closer physical distance – for example, at the cell level. We term this *per-cell* integration. In this level of integration, each hybrid memory cell would actually be composed of multiple cells – one for each different memory type. Because each cell consists of co-located memory types, data only has to move an extremely short distance. This close integration comes with fabrication challenges, but can provide enormous advantages in lower latency and lower energy costs for data movement between memory types.

On the other side of the spectrum, it is possible to construct a hybrid memory design using different physical memory chips. This *per-chip* level of integration is the easiest to fabricate, but data movement when going off-chip is extremely expensive and normally more than negates any advantage to using a hybrid memory design. However, it is included in our classification system for completeness.

Another design point in the level of integration exists between per-chip and per-array. With 3D chip stacking technologies, different memory arrays can be fabricated on separate substrates, and then vertically stacked on top of one another for close phpsical proximity. This design point is easier to fabricate than per-cell integrated hybrid memories. Notably, it is arguably easier than per-

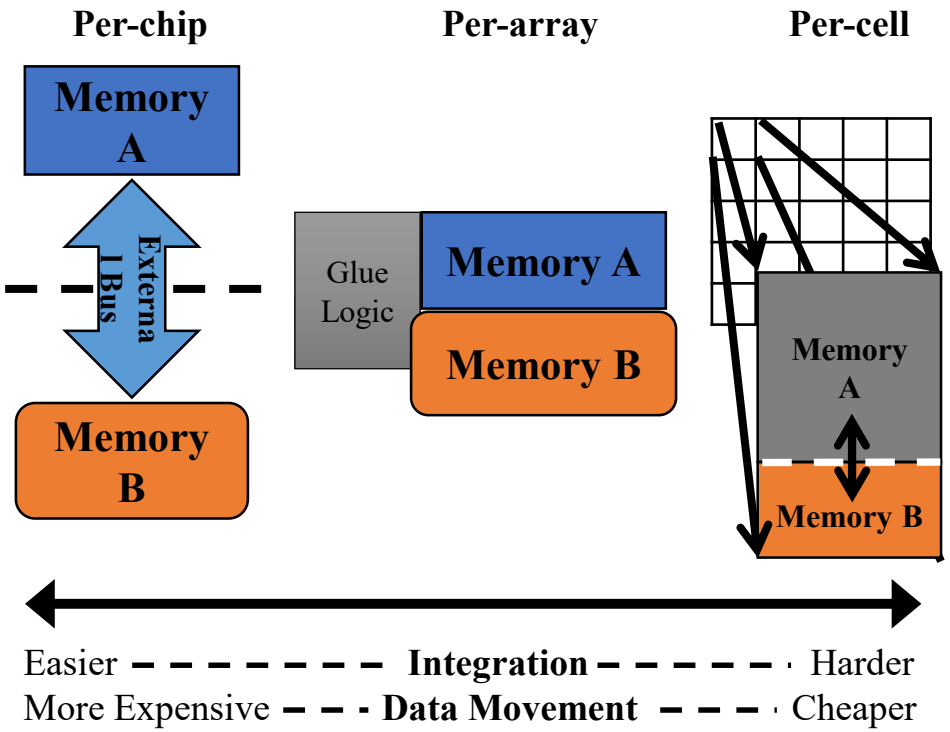


Figure 2.3: Level of integration spectrum for hybrid memory design.

array fabrication, which requires two memory technologies on the same substrate, while providing much of the benefit of the closeness of per-array fabrication via 3D stacking.

Figure 2.3 visualizes these design points on a line. We can think of distance along the line indicating physical closeness of the different memory cells. At one end is per-cell integration, with the closest physical distance between different memories. As the line progresses, physical distance increases between the memories. We first come to per-array, then 3d stacked arrays, and lastly per-chip. On the other hand, while following the line, we start with a very high difficulty in fabrication, and as the line progresses from per-cell to per-chip, fabrication difficulty decreases.

2.4 Trade-Offs

Architects consider certain trade-offs when designing hybrid memory systems. For example, when deciding on a serial versus a parallel organization, an architect must decide whether the additional access control offered by a parallel organized hybrid memory design is worth the additional wires, routing difficulty, and area cost over a serially organized one.

A similar trade-off is inherent in deciding whether or not to expose the hybrid design to the outside. An externally visible hybrid memory allows possible optimization from software or external hardware, but requires more ports and wires to connect each memory type to the outside. Hiding the hybrid design makes the design easier to use as a drop-in replacement for an existing memory structure.

When choosing level of integration, the most obvious trade-off is in fabrication difficulty versus ease of data movement. A more closely integrated hybrid memory design is harder to fabricate, but offers cheaper inter-memory data movement. For applications where the cheaper movement makes a larger difference, it may be worth the extra overhead in production.

2.5 Hybrid Memory Related Work

With the classification system explained above, we can group some examples of hybrid memory proposals.

The first two parameters, internal organization and external view, describe

how the memory operates from the system functionality point-of-view. The third parameter describes particular physical tradeoffs, such as energy and latency.

In this section we have grouped selected recent works by internal organization and external view.

2.5.1 Serially Organized Memories

These hybrid memories are organized with their memories in a serial path, and interface with the outside via one memory only. This organization is often used when one memory caches the other.

All the following proposals present themselves as one memory structure – they hide the complexity of the hybrid memory to the system.

2.5.2 Parallel Organized Memories

Parallel organized memories can be seen by the outside system as one or more memories. We will denote these categories hidden and exposed.

Internally, a parallel organization means that the memory controller can pull data from both type of memory in the proposed structure.

Examples of Serially Organized Memory Designs		
Year	**Authors**	**Title and Description**
2009	Qureshi et al. [37]	**Scalable High Performance Main Memory system Using Phase-Change Memory Technology** Main memory is replaced with Phase-Change Memory, using DRAM as a cache.
2011	Bheda et al. [6]	**Energy Efficient Phase Change Memory-based Main Memory for Future High Performance Systems** Main memory is replaced with Phase-Change Memory, using DRAM as a cache.
2013	Kvatinsky et al. [24]	**Memrister-Based Multithreading** Flip-flops use co-located memristors to give them fast non-volatile operations.

Table 2.1: Serially organized memories.

2.6 Conclusion

In this section we describe a broad classification system to place hybrid memory work in context with other work in the field. We use three criteria: internal organization, external view, and level of integration to classify hybrid memory proposals. We show how selected works fit into the classification scheme. It is hoped that architects will use the system to gain insight into how proposals conforming to certain criteria relate to other proposals with different criteria.

Examples of Parallel Organized Memory Designs		
Hidden from external view		
Year	**Authors**	**Title and Description**
2009	Sun, G. et al. [42]	**A Novel Architecture of the 3D-stacked MRAM L2 Cache for CMPs** A per-array hybrid design uses SRAM for write-heavy data, and a MRAM for everything else.
2011	Sun, Z. et al. [43]	**Multi Retention Level STT-RAM Cache Designs with a Dynamic Refresh Scheme** Uses STT-RAMs with varying retention times in one level of cache to provide fast writes, long retention times and uses less refresh power.
2011	Ghasemi et al. [14]	**Low-voltage On-Chip Cache Architecture Using Heterogeneous Cell Sizes for High-Performance Processors** Uses differently sized SRAM cells for cache, disabling the less reliable sizes when running at lower voltages. Uses entire cache and all SRAM cell sizes when the processor is running at high performance.
2013	Lee et al. [25]	**Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture** Make DRAM have two tiers of latency by adding an isolation transistor to ease bitline load. The faster DRAM section can be treated as a hardware cache for the slower DRAM section (hidden externally), or both can be exposed to the OS and the OS can manage what data goes where (exposed externally).
Exposed to external view		
2011	Yang et al. [57]	**I-CASH: Intelligently Coupled Array of SSD and HDD** Uses software to combine SSD and HDD into one system (managed by the OS), which writes to HDD and reads from SSD.
2013	Li et al. [26]	**Hybrid CMOS-TFET-based Register Files for Energy-Efficient GPGPUs** Uses TFET arrays for read-only registers, keeping CMOS arrays for everything else.

Table 2.2: Parallel organized memories.

CHAPTER 3

HYBRID SRAM-FLASH FOR NON-VOLATILE COMPUTING

My first project investigated the potential benefits of a cell-level integrated hybrid memory: merged Flash and SRAM. This combination was undertaken in order to reduce the cost and time for an architectural level, non-volatile save and restore. These operations are crucial for non-volatile computers in power-constrained environments, which are systems that experience frequent power outages. They rely on non-volatile operations to quickly snapshot existing computation before a sudden power loss, and to quickly restore that computation upon power restoration. If these operations are expensive, then non-volatile computers would not be feasible. This hybrid memory design also benefits general purpose processors not limited by power outages. General purpose processors have complex power management and frequently transition in and out of sleep states. Saving architectural state quickly and cheaply enables the quick restoration of work on later wake-up, and more fine-grained transitions into sleep and awake states.

Saving architectural state from volatile memory on-chip to non-volatile memory off-chip is a time and energy intensive operation. A cheaper and faster non-volatile save and restore can enable a computing device to spend more time in deep sleep, lengthening deployment times and battery life. On the other hand, the device could instead use the energy saved for other computational purposes.

Most of the energy and time cost of saving state comes from the movement of data off-chip to memory, and then in the energy and time required to write the non-volatile memory. By co-locating the non-volatile memory next to existing

memory, the movement costs can be eliminated. The costs of saving data to non-volatile memory remain the same.

To eliminate data movement costs, one solution would be to replace the entire memory state with non-volatile memory. This is not ideal due to the high energy cost of non-volatile memory read and write operations compared to volatile SRAM. In particular, non-volatile memory requires high write energy. Using non-volatile memory as working memory would quickly prove inefficient, as any savings gained from not moving data would be spent in writing. Another important consideration is the endurance of non-volatile memory. Many non-volatile technologies are limited in the number of writes they are capable of supporting. Use as frequently written state elements may wear out the memory, limiting the device's lifetime.

In this chapter, we investigate a non-volatile processor architecture that deeply integrates non-volatile memory with volatile memory such as SRAMs at the cell level. Instead of relying on two separate arrays of volatile and non-volatile memory, we create hybrid SRAM and flip-flop cells where a non-volatile transistor is incorporated in each memory cell. For normal operations, this hybrid design allows the memory cells to operate as regular volatile state elements with comparable latency and energy consumption. The hybrid design also enables state to be quickly stored in and restored from non-volatile memory when necessary because data movements between volatile and non-volatile elements happen within each cell. Moreover, the hybrid memory can perform non-volatile operations on many memory cells in parallel without interrupting normal processing operations.

To fully realize the potential of non-volatile computing, we design new hy-

brid memory cells based on floating-gate transistors (Flash memory), described in 3.3, to investigate system-level integration issues through a non-volatile microcontroller prototype, and present comprehensive evaluations of both the memory cells and the non-volatile microcontroller. The new non-volatile memory cells significantly reduce the energy consumption on non-volatile operations compared to other non-volatile cell designs that require high currents to program (such as phase-change memory). The design of the non-volatile processor also decouples energy intensive operations such as a program voltage generation from non-volatile operations by performing them at different times in order to reduce instantaneous load on the power source. The new non-volatile memory and architecture allow processor state to be saved in non-volatile elements even after a power failure without resorting to periodic checkpointing in order to preserve existing computation.

The proposed non-volatile SRAM and flip-flop designs were implemented as transistor-level models and studied with SPICE simulations based on parameters obtained from the IBM 65nm transistor model and experimental measurements of floating-gate transistors.

Our simulation results confirm the non-volatile capability of the state elements and suggest that the overheads are quite low. For regular operations, the non-volatile SRAM and flip-flop show small increases in delay (~18-38%) and energy (15-25%). The non-volatile store operations are also quite efficient, taking less than 10μs and consuming practically no energy within a memory cell. In fact, the majority of the energy is consumed by a charge pump to raise a voltage at a floating gate. To study the architecture level integration issues, we also built a transistor-level model of an 8-bit microcontroller and studied

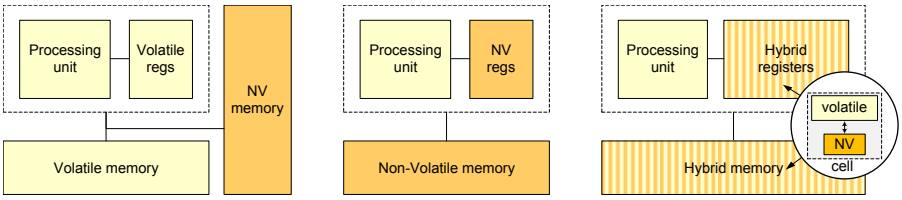
its performance with and without the non-volatile capability using the Synopsys HSIM co-simulation environment. The results suggest that the system-level overheads are even lower for regular operations, only incurring a few percent (1-3%) overhead in energy without an impact on the operating clock frequency. The non-volatile checkpoint including the overhead of a charge pump only consumes 172 pJ.

The rest of the chapter is organized as follows. Section 3.1 defines non-volatile computers and discusses challenges. Then, Section 3.2 discusses main applications of the proposed architecture. Section 3.3 introduces non-volatile building blocks, namely, the hybrid SRAM and flip-flip cells. Section 3.4 shows our prototype non-volatile microcontroller, with the new non-volatile elements and discusses the issues that are related to integrating the non-volatile cells in a processor. Section 3.5 presents experimental results for both our non-volatile memory cells and the microcontroller implementation. Finally, Section 3.6 discusses the related work and Section 3.7 concludes.

3.1 Non-Volatile Computing

In this section, we discuss high-level architecture options and challenges in designing non-volatile computing systems. Here, we use the term *non-volatile computers* to refer to systems that can almost instantly save their state in a non-volatile fashion so that operation can continue even across an unanticipated power interruption.

In general, a non-volatile computer needs to be able to perform three non-volatile (NV) operations in addition to regular computations: `NV store`, `NV`



(a) Discrete NV architecture.

(b) Full NV architecture.

(c) Hybrid NV architecture.

Figure 3.1: Architecture options for systems with non-volatile capabilities.

`restore`, and `NV erase`. Before a power interruption, a non-volatile system must be able to store its state into non-volatile storage. To resume its operation, a system restores the state from the non-volatile storage. Finally, non-volatile memory technologies typically require an "erase" operation to reset its state before being programmed with new values. How these three operations are supported largely determines the non-volatile computing architecture.

3.1.1 Architecture Trade-offs

Figure 3.1(a) shows a discrete NV architecture that is commonly used today to incorporate non-volatile memory into a computing system, where a non-volatile memory array is added as a separate module to a volatile system, either on-chip or off-chip. This architecture style has virtually no impact on the performance of regular computations and is well-suited for infrequent NV checkpoints. However, `NV store` and `NV restore` operations are expensive in terms of both latency and energy consumption because data must be moved sequentially between discrete modules. As a result, this discrete architecture cannot effectively handle unexpected power failures or support applications that require a quick response time.

One possible approach to enable fast non-volatile checkpointing is to re-

place all volatile elements with non-volatile ones as shown in Figure 3.1(b). Effectively, this fully non-volatile architecture eliminates the need for explicit `NV store` and `NV restore` operations by storing everything in a non-volatile fashion in the first place. However, in this architecture, regular computations are directly impacted by the limitations of non-volatile memory technologies, whose operations are often much slower and energy intensive compared to typical volatile memory elements. Moreover, the full NV architecture may simply be impractical for non-volatile memory technologies with limited endurance because every processor operation wears out memory.

We integrate non-volatile elements into volatile memory at each bit granularity. As shown in Figure 3.1(c), in this architecture, we will be building registers and memory arrays with our hybrid state elements (SRAM, flip-flops, etc.) that contain both volatile and non-volatile storages within each bit cell. For normal computation operations, the architecture relies on the volatile storage to provide the performance that is close to traditional volatile computers. The `NV store` and `NV restore` operations on the architecture are explicit but can be almost instantaneous with a low energy requirement because the data movement between volatile and non-volatile elements can be performed for many (possibly, all) cells in parallel, and within each cell. Table 3.1 summarizes the characteristics of each NV architecture style.

3.1.2 Challenges of the Hybrid NV Architecture

While conceptually simple, the hybrid NV architecture presents new challenges and questions that need to be answered in order to fully realize its potential.

	Normal Op Impact	*NV Store/Restore*
Discrete	Negligible	Explicit, sequential, high energy
Full NV	High	Implicit on every operation
Hybrid	Low	Explicit, parallel, low energy
	NV Erase	*NV Area*
Discrete	Explicit, background	Compact
Full NV	Implicit	Compact
Hybrid	Explicit, background	Less compact

Table 3.1: Trade-offs in the non-volatile architecture design styles.

NV power and energy consumption. The hybrid state elements have the potential capability to create an instant checkpoint of the entire system state by locally copying the state to a non-volatile element within each bit cell. However, such a parallel copy operation implies that a large number of non-volatile writes need to be performed simultaneously and poses a challenge in terms of handling power consumption.

Integration overheads. The integration of non-volatile components into standard volatile state elements such as SRAM and flip-flops may introduce overheads in common volatile read/write operations. For example, the additional components increase the capacitance load within each memory cell and may also result in longer wires due to an increase in size. The `NV erase` operation may also interfere with regular operations.

Endurance. Many non-volatile memory technologies have limited endurance; memory cells may not work reliably beyond a certain number of program/erase cycles. To avoid wearing out its non-volatile parts, the proposed hybrid NV architecture has the `NV store` and `NV erase` operations explicitly performed only when non-volatility is needed.

Fabrication. In practice, one major obstacle in realizing any non-volatile architecture is today's separation between logic and non-volatile memory fabrication processes. However, there is no technical reason why both processes cannot be tightly integrated. In fact, Freescale uses nanocrystal flash memory (the type of flash that we use) on microcontrollers in volume production [13]. Foundries such as TSMC have also started supporting embedded (on-chip) flash memory.

3.2 Applications

A microprocessor that can almost instantly checkpoint its state in a non-volatile fashion can benefit a broad range of application scenarios.

3.2.1 Continuous Computation under Unstable Power Sources

Many of today's mobile and embedded computing devices rely on a battery. However, batteries may not be an option in deeply embedded devices due to their limited lifetime or environmental concerns as in bio-implanted devices. In such cases, a computing device needs to harvest energy from its environment via sources such as radio-frequency signals, solar energy, vibration, and thermal differences.

Self-powered devices could potentially operate perpetually and autonomously without human intervention. However, such devices face a challenge of instability in its power source. Moreover, most energy harvesting techniques cannot provide enough power even for ultra low-power microcontrollers [8]. As a result, self-powered devices must deal with relatively frequent and

unpredictable power failures, and are often only capable of operating intermittently.

For a traditional computing device, an unpredictable loss of power results in loss of its state in volatile elements such as SRAM and flip-flops. As a result, today's self-powered devices are limited to very simple computations such as just collecting sensor inputs and sending them out. If a computing device can maintain its state across a power failure and continue its computation, such a capability will significantly broaden possible uses of self-powered devices by enabling long computations.

3.2.2 Idle Power Reduction with Fast Response Time

Mobile and embedded devices often spend most of their time idling. For example, sensor network nodes may only be active periodically to collect and process data. For such devices, static leakage represents a significant component of the overall energy consumption.

The architecture based on the hybrid non-volatile state elements presents an attractive solution to reduce idle power consumption without sacrificing response time. The proposed non-volatile architecture can completely turn off its memory while idle and almost instantaneously restore its state on a wake-up.

3.3 Non-Volatile State Elements

For non-volatile computing, we developed a hybrid SRAM (NV-SRAM) and flip-flop (NV-DFF). These hybrid state elements incorporate non-volatile floating-gate transistors directly into each bit-cell so that their state can be locally copied between volatile and non-volatile parts without reading data out of each memory cell.

While there exist several non-volatile memory technologies that can potentially be used in non-volatile architectures, we use floating-gate Flash memory as the baseline non-volatile technology. Flash memory is arguably the most mature and widely adopted form of non-volatile memory today and more importantly allows the hybrid cells to be optimized for low power NV operations.

Two-terminal devices such as MRAM, RRAM, and PCRAM rely on a high current pulse to change their state, with a noticeable energy consumption on each program or erase operation. This characteristic presents a particular challenge in handling power failures where a large number of bits need to be programmed at once with a limited amount of energy. On the other hand, the floating-gate device uses a gate voltage to change its non-volatile state and only requires a trace amount of current. As a result, the power consumption on a floating-gate device comes from charging a small gate capacitance, and from generating a program/erase voltage, which can be done before a power interruption on a global dynamic node.

Table 3.2 shows the estimated power consumption on the `NV store` operation for hybrid SRAM designs with various types of memory technologies. The numbers are estimated by scaling published results of each cell to a 70 nm node.

SRAM Design	Power (μW)
ReRAM [51]	24.6
PCRAM [44]	378
MRAM [39]	32.2
FeRAM [32]	0.124
NC Flash (this work)	0.017

Table 3.2: Estimated power dissipation for a `NV store` operation on various types of hybrid SRAM designs. The power is estimated per cell at a 70 nm node.

The estimation for the NV Flash technology that is used in this paper includes the power consumption of a charge pump that generates program and erase voltages. The results demonstrate the potential benefit of using the floating-gate non-volatile device in hybrid SRAM designs.

Our hybrid SRAM and DFF integrate 2 non-volatile NMOS floating-gate transistors into standard SRAM and flip-flop cells. The cells need two additional signals in order to control connections between volatile and non-volatile parts (`EN`) and to program/erase floating-gate transistors (`PE`). The hybrid memory can perform three non-volatile operations, `NV store`, `NV restore`, and `NV erase`, in addition to typical volatile read and write operations. The `NV erase` operation can be performed in the background overlapping with volatile accesses.

The details of our hybrid SRAM and DFF are discussed next. They are based on a previous work [38].

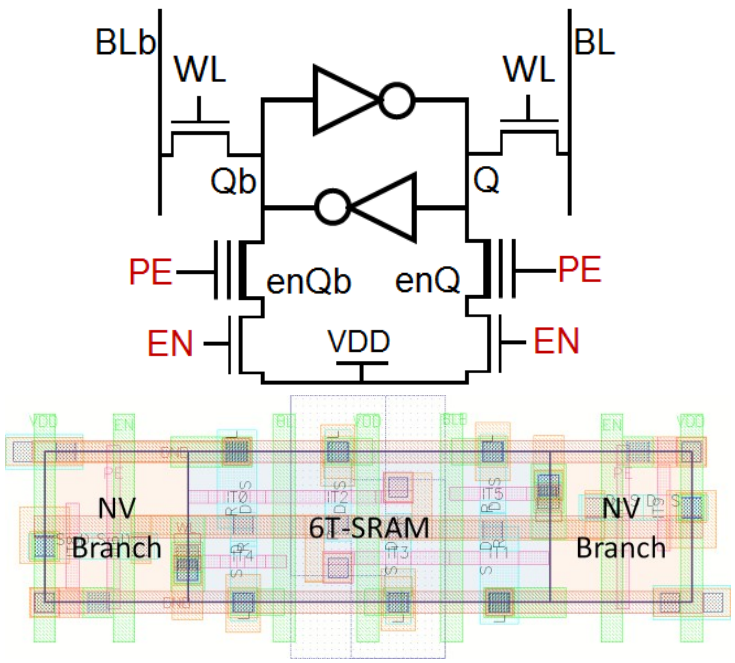


Figure 3.2: Hybrid SRAM-Flash cell circuit and layout.

3.3.1 Hybrid SRAM Cell (NV-SRAM)

The hybrid SRAM-Flash cell, hereafter referred to as NV-SRAM, consists of a standard 6T SRAM augmented with 2 non-volatile NMOS floating-gate transistors as shown in Figure 3.2. The floating-gate transistors are connected directly to the `Q` and `Qb` nodes of the SRAM cell, respectively. Enable transistors connect the floating-gate transistors to VDD. The enable transistors are controlled by signal `EN`, and the non-volatile program/erase operations are controlled by signal `PE`. In the following discussion, we use 65nm transistor models operating at 1.2V to describe detailed operations.

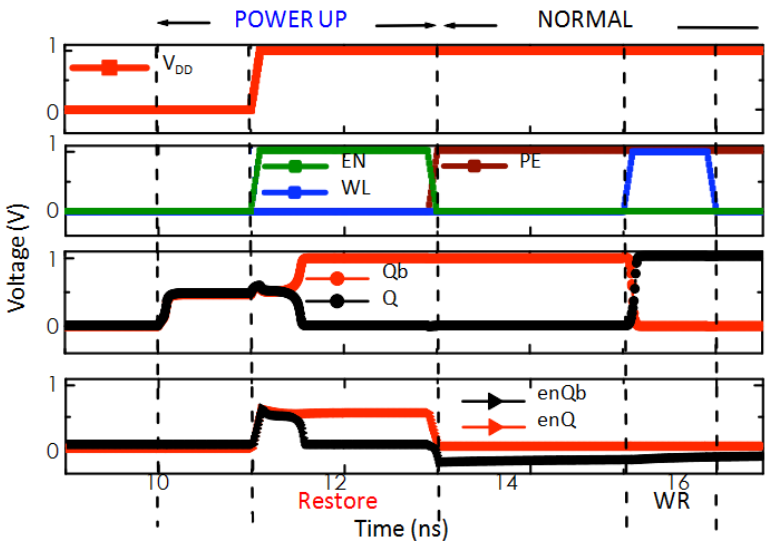
At a high level, the hybrid SRAM cell can perform three non-volatile operations, `NV store`, `NV restore`, and `NV erase`, in addition to typical volatile read and write operations in the regular mode. For regular volatile operations, `EN` is off and `PE` is off (0V for both signals). In this case, the cell is essentially a

normal 6T SRAM with small extra capacitance on internal nodes Q and Qb.

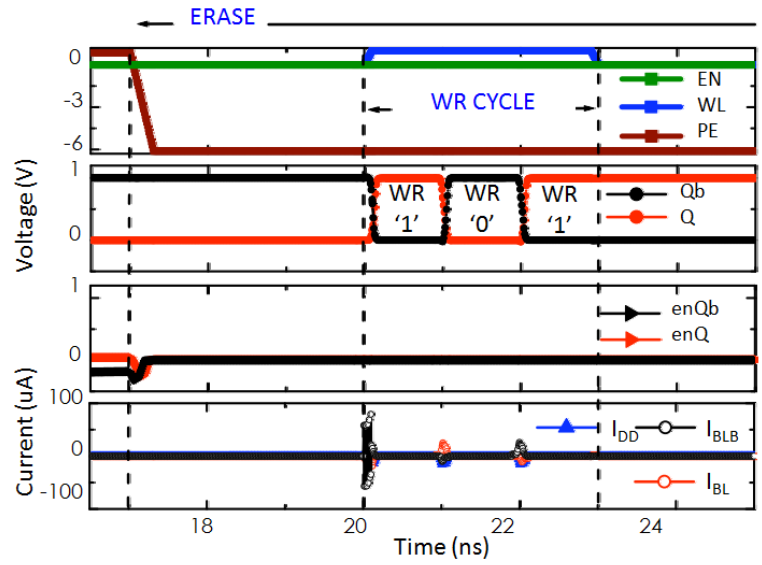
For the NV store operation, the wordline access transistors are first turned off (WL is set to 0V), isolating the cell from the bit lines. The EN signal remains off, which isolates the source of each floating-gate transistors from VDD. The program voltage (6V) is then applied to the gate of the floating-gate transistors (PE) so that electrons tunnel into the floating gate. Since the transistor turns on, the voltage in the channel of the floating-gate transistor is now influenced by the connected state node. The tunneling current is exponentially dependent on the voltage difference between the channel and gate. As a result, the floating-gate transistor with a lower channel bias sees an exponentially higher electron tunneling rate than the other device with a higher channel bias. When the threshold voltage (Vth) difference between the two floating-gate devices becomes sufficiently large, the store operation is complete. Even with the worst-case process variations, our SPICE simulation results suggest that 10 μs is sufficient. Note that because the enable transistor is off, there is no current drained.

To restore the SRAM state after a power interruption, we use the sequence of signals in Figure 3.3(a). While the wordline transistors remain off, the cross-coupled inverters are turned on, while a resistive path from Q and Qb to VDD is created via the floating-gate transistors by asserting EN. The resistance of node Q and Qb to VDD is determined by the leakage through their associated floating-gate transistors. The programmed flash has a higher Vth and higher resistance to VDD. This asymmetry forces the volatile SRAM cell to restore its state after a power interruption.

The non-volatile transistors need to be erased to reset their threshold voltages (Vth) to the 'non-programmed' state before they can be reliably pro-



(a) NV restore sequence.



(b) NV erase sequence.

Figure 3.3: The `NV restore` and `NV erase` sequences for the hybrid SRAM-Flash cell.

grammed again. Figure 3.3(b) shows the sequence for the `NV erase` operation. To erase the devices, a large negative voltage (-6V) is applied to the `PE` signal so that electrons are pushed off the floating gate. The `NV erase` operation can be performed in the background. Regular volatile operations are not affected by

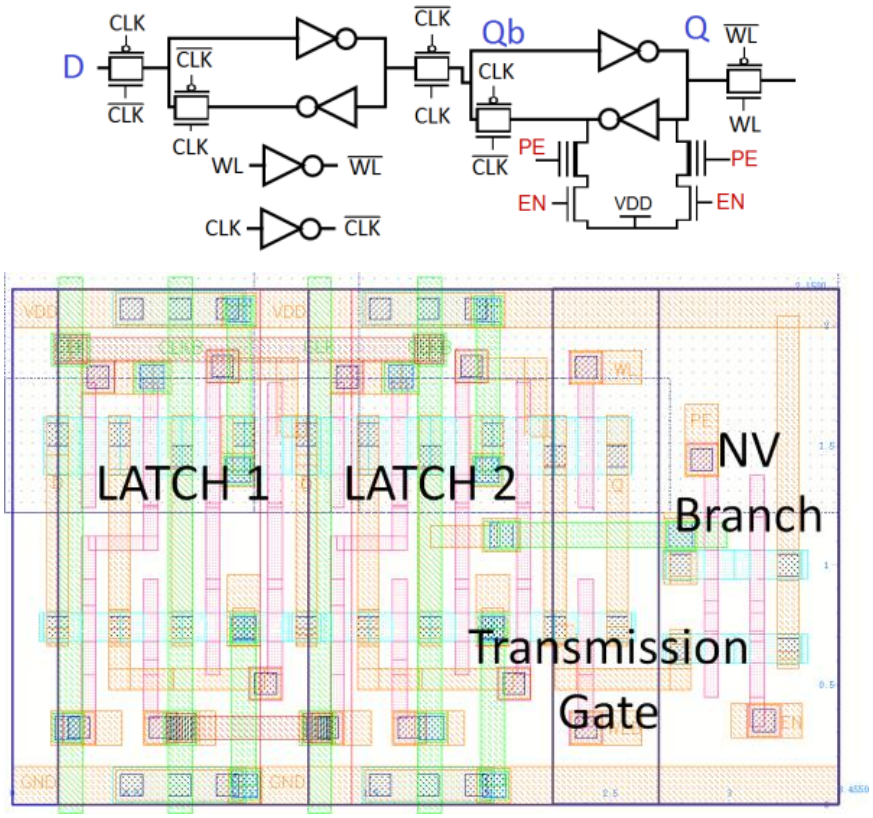


Figure 3.4: Hybrid SRAM-Flash D flip-flop circuit.

the erase operations. As an example, the figure shows volatile write operations that are overlapped with the `NV erase` operation.

3.3.2 Hybrid Flip-Flop (NV-DFF)

Figure 3.4 shows our hybrid D flip-flop design, which augments the slave latch with non-volatile transistors. Similar to the hybrid SRAM, the hybrid flip-flop supports regular volatile operations and three non-volatile operations. For regular volatile operations, the `EN` signal is turned off so that the floating-gate tran-

sistors are isolated from VDD. In this case, the flip-flop operations are identical to standard volatile flip-flops. A transmission gate is also added to the output of the flip-flop, used only to isolate the slave latch during `NV restore`, as extra capacitance on the normally attached fanout could bias one particular side of the latch and hence the state.

The `NV store` operation is similar to the hybrid SRAM. First, the clock signal (`CLK`) is held at 0 to halt flip-flop operation. At this point, the flip-flop state is stored on the slave latch nodes `Q` and `Qb`. Then, the programming voltage is applied to the floating-gate transistors. As in the SRAM, the `Q` and `Qb` node voltages are expressed at the floating-gate transistor source, drain, and channel. The difference in the voltage between the gate and the channel causes the two flash devices to be programmed differently, storing the state of the flip-flop.

The `NV restore` operation for the flip-flop is similar to the hybrid SRAM. The transmission gate at the output is turned off to isolate the latch. The `EN` signal is turned on, connecting `Q` and `Qb` to VDD via the floating-gate transistors. Then the cross-coupled inverters in the slave latch are powered on. The asymmetry in the leakage current restores the flip-flop state.

The `NV erase` operation is identical to the SRAM case. With `EN` off, a high negative voltage is applied to `PE` to restore the threshold voltages for both floating-gate transistors. The erase operation can take place in the background.

3.4 Non-volatile Microcontroller Prototype

In this section, we describe a prototype design of a non-volatile microcontroller that uses the hybrid state elements.

3.4.1 Baseline Architecture

In this study, we use a simple 8-bit microcontroller as the baseline architecture. Our prototype implementation is based on an open-source clone [21] of the Xilinx PicoBlaze microcontroller. Small microcontrollers are widely used in embedded devices in energy-constrained environments where the non-volatile capability will be particularly useful.

The processing core of the baseline microcontroller consists of an ALU, 16 8-bit general-purpose registers (GPRs), and a set of other registers such as a 10-bit program counter (PC), an 17-bit instruction register, and 3 condition codes. The processing core is connected to three types of on-chip memory modules. For program instructions, there is a non-volatile flash, which can store up to 1024 instructions (1.7 KB). For dynamic data, the microcontroller includes a 32-entry stack to store return addresses and a 64-B scratch pad. There is no off-chip memory. The microcontroller is not pipelined and executes one instruction every two clock cycles. This architecture configuration is comparable to today's low-end commercial microcontrollers[1].

[1] Atmel's 8-bit microcontroller ATtiny4 contains a 512-B program flash, a 32-B SRAM scratch pad, and 16 8-bit GPRs [3].

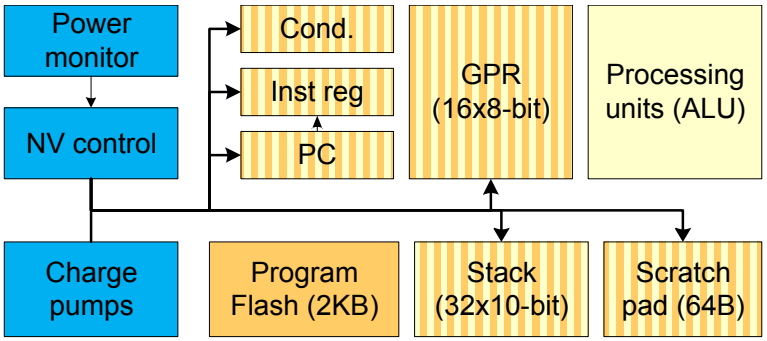


Figure 3.5: Non-volatile architecture modifications.

3.4.2 Changes for Non-Volatile Computing

The non-volatile architecture replaces volatile registers and memory modules in a traditional system, whose content needs to be preserved over a power interruption, with the hybrid non-volatile elements and adds a set of control and support structures to manage non-volatile operations.

Figure 3.5 illustrates the non-volatile microcontroller architecture. As shown with the striped boxes, all volatile state elements (registers, stack, and scratch pad) now use the hybrid memory. The ALU and the program flash remain unchanged. For our microcontroller, virtually all state elements are architecturally visible and need to be preserved on a power interruption. More complex microprocessors with advanced techniques such as speculation could preserve only architectural state.

To support and manage the new hybrid state elements, the non-volatile architecture needs three additional components, which are shown as blue (dark) boxes. The floating-gate transistors that we use in the non-volatile elements require program and erase voltages that are higher than the operating voltage of the microcontroller. Therefore, the microcontroller needs a charge pump to generate these voltages, which are common for all hybrid state elements.

The non-volatile (NV) control module provides control signals to manage the hybrid state elements as well as the charge pump. On a power interruption, the control module initiates the `NV store` operation to checkpoint the microcontroller state. Then, when the power is restored, the module performs the `NV restore` operation to restore the state from the floating-gate transistors. Once the microcontroller resumes its normal operation, the control module performs the `NV erase` operation in the background to be ready for the next power interruption. Because a power recycle does not 'reset' a microcontroller, a separate hard reset is needed to clear the architecture state.

For the NV control module to know when the power is interrupted or restored, a power monitoring circuit signals the control module when the supply voltage is below or above certain threshold levels.

3.4.3 Optimizations

Charge Pump Power Consumption

Our hybrid state elements are designed to minimize their power consumption on non-volatile program (store) and erase operations. In fact, power consumption of non-volatile operations comes mostly from generating high voltages and delivering them.

For applications whose goal is to minimize the total energy consumption, a natural organization is to only turn on the charge pump on a program or erase operation when a high voltage is required. However, in the context of handling unpredictable power interruptions, this organization implies that the

charge pump consumes energy after a power interruption is detected. Instead, for such scenarios, we design the architecture to focus on minimizing the energy consumption after a power failure by pre-generating a program voltage. The charge pump is turned on soon after a power-up to generate a program voltage. Then, the charge pump operates with a slow clock to simply maintain the voltage level by compensating a small amount of leakage. This approach enables the actual programming of non-volatile cells to be performed with minimal energy consumption.

NV Erase Operation

A typical flash memory controller performs an erase operation just before a program operation. This design decision is largely inevitable because the typical systems require the flash content to be maintained until they are explicitly updated. However, combining the erase and program operations has an undesirable consequence of paying the latency and energy overheads of the erase operation at the time of storing a new value into flash.

Fortunately, in the context of non-volatile computing, the non-volatile elements need to retain their state only during a power interruption. Therefore, the `NV erase` operation can be performed anytime after the power is restored and before the next power failure. In our prototype, the NV control module initiates the `NV erase` operation in the background right after the microcontroller resumes its execution.

Control Signals

To amortize the overheads of additional control signals, our prototype organizes the hybrid state elements, which are physically located closely together, into a small number of groups. Fortunately, most processors already organize architecture state elements into arrays to amortize the cost of peripheral circuits.

3.5 Evaluation

This section presents evaluation results for our non-volatile microcontroller prototype as well as the hybrid SRAM and flip-flop cells. We also validated the non-volatile functionality through SPICE simulations. Here we focus on area, performance, and power/energy consumption.

3.5.1 Methodology

To verify functionality and evaluate the overheads of the proposed hybrid SRAM (NV-SRAM) and flip-flops (NV-DFF), we created transistor-level models, performed layout, and extracted parasitics for both baseline cells and our hybrid SRAM and flip-flop cells with floating-gate transistors. The state elements are simulated with HSIM using IBM 65nm low-power transistor BSIM4 models.

For the microcontroller prototype, we use a standard-cell ASIC flow along with the Synopsys HSIM co-simulation environment to study its functionality and overheads. First, the prototype microcontroller in Verilog HDL is synthe-

Operation	Volatile (J)	Hybrid (J)	Diff
SRAM Read	4.056×10^{-15}	4.836×10^{-15}	19.23%
SRAM Write	6.228×10^{-15}	7.332×10^{-15}	17.73%
DFF 0-to-1 transition	3.096×10^{-15}	3.876×10^{-15}	25.19%
DFF 1-to-0 transition	5.660×10^{-15}	6.493×10^{-15}	14.71%

Table 3.3: Energy overheads of hybrid state elements.

sized with Synopsys Design Compiler (DC) and a commercial 65nm standard cell library to produce a structural gate-level netlist. The synthesized design includes the processing core (ALU, registers, and control) and the stack and scratch pad modules. The program flash is simply considered as a black box. Given that all memory modules are quite small we use flip-flops (DFF) for memory. In order to compare costs of non-volatile extensions, we replaced the DFF cell in the library with our own DFF design and layout; a volatile one for the baseline and the non-volatile version for the NV microcontroller. Then, baseline and non-volatile designs are placed and routed using Cadence Encounter, which provides estimates for area, clock frequency, wire loads, etc.

3.5.2 Non-Volatile State Elements

Regular Operations

The hybrid state elements include additional access and floating-gate transistors compared to standard SRAM and flip-flop cells. These additional transistors increase capacitance loads. The increased cell area also results in longer bitlines and wordlines. As a result, the hybrid SRAM and flip-flop cells consume more energy and have longer delays compared to the baseline cells.

Operation	Volatile (ps)	Hybrid (ps)	Diff
SRAM Read	56.438	66.555	17.93%
SRAM Write	82.289	95.288	15.80%
DFF 0-to-1 transition	68.479	87.901	28.36%
DFF 1-to-0 transition	57.564	79.359	37.86%

Table 3.4: Performance (delay) impact of hybrid state elements.

Table 3.3 compares the energy consumption of the baseline and the hybrid cells for both SRAM and flip-flops. For SRAM cells, the table shows the energy that is required to read and write, respectively. Bitline lengths and wordline lengths are scaled appropriately, approximating performance in an array. For D flip-flop cells, the table shows the energy that is consumed on each data transition. Similarly, Table 3.4 shows the performance in terms of a delay to switch state. SRAM delay includes array bitline and wordline increases from the larger area of the NV-SRAM.

Non-Volatile Operations

For the `NV store` operation, experiments show that 10μs at 6V is sufficient under the worst-case process variations (0.1V threshold and 30% area). The `NV erase` operation is a mirror image of `NV store` and takes about the same amount of time (10μs at -6V). The `NV restore` operation is fast; 2ns was enough to restore state from power-down. The simulations confirm that power consumption during the non-volatile operations is negligible. Both hybrid SRAM and flip-flop are virtually identical in terms of their non-volatile operations.

Inst Class	Volatile (pJ)	Hybrid (pJ)	Diff
Computation	5.39	5.47	1.38%
Load/Store	5.11	5.19	1.54%
Control Flow	4.72	4.80	1.63%

Table 3.5: Per-instruction energy consumption.

Area

The area overheads of the hybrid SRAM and flip-flop cells are evaluated by careful layout according to conservative 65nm design rules. For the SRAM cell, we add 4 minimum size transistors to a normal 6T cell, which results in a 63% overhead. The flip-flop design introduces 6 additional transistors to the baseline with 22 transistors. Our layout shows a 40% area overhead.

3.5.3 System-Level Overheads

Regular Operations

Table 3.5 compares the energy consumption between the baseline and the non-volatile microcontrollers for each class of instruction. The estimate is an average of all instructions in the class; each instruction has 20 different runs with randomly selected input values. The results show that the energy overheads on regular computations are minimal. The system-level overheads are quite low because an instruction execution involves volatile modules whose energy consumption is not directly affected by the non-volatile architecture. The small overhead is due to the slightly larger area of the volatile cells, which adds wire capacitance. We also note that the energy consumption in the table does not

include the energy to read an instruction from the program flash.

In the experiments, we ran the microcontroller at 10MHz. Both baseline and non-volatile implementations could easily run at a much higher frequency. As a result, the additional propagation delay of the hybrid cells did not have any impact on the performance of the overall system.

Non-Volatile Operations

Our non-volatile microcontroller integrates 989 hybrid flip-flops into one chip. Because they can be performed simultaneously, the `NV store` and `NV erase` operations take 10μs, and the `NV restore` can be done in a few nanoseconds as in the single-cell case.

To estimate the system-level energy consumption, we implemented a SPICE model of a charge pump based on a published design for flash memory [12]. The charge pump was connected to the gates of 1,978 NV transistors through a switch, and the wire capacitance was estimated from Cadence Encounter after the place and route and added to the netlist. The storage capacitor on the output is sized to be 1.5x the total capacitance of the NV transistor gates and wire network. Initially, while disconnected from NV transistor gates (the switch is open), the charge pump operates at 120MHz and brings the voltage from 0 to about 10V. This initial charging takes about 3μs and 160 pJ. Most of energy and time on this step is spent to charge up the wire capacitance. Then, the charge pump switches off and only turns on to maintain the output voltage when needed. In this mode, the charge pump consumes about 1.2μW to compensate for leakage currents. Finally, during a NV program step, the charge

	Baseline (μm^2)	NV (μm^2)	Diff (%)
Total	67,192	80,587	19.9
- Scratch pad	40%	-	-
- Stack	25%	-	-
- Processing core	35%	-	-

Table 3.6: Area estimates for baseline and NV microcontrollers. Estimates include processing core and data memory, but not the program flash.

pump and its storage capacitor is directly connected to the NV transistors. Including the leakage current of 1.2μW, the program operation itself consumes 12 pJ. Overall, the `NV store` operation takes about 172 pJ to program 989 DFFs (0.174 pJ per DFF).

Area

Table 3.6 summarizes the chip-level area estimates for both baseline and non-volatile microcontrollers. The area includes the processing core, the stack, and the scratch pad, but not the program flash. In a rough estimate, the scratch pad and the stack account for 40% and 25% of the design, respectively. The overall area overhead is only about 20% with NV flip-flops. We believe the area overhead of SRAM-based implementation will be around 25%.

Lifetime

In the hybrid NV architecture we study, a checkpoint only occurs on power-down. Using the 10^{12} P/E cycle limit of our flash cell, the system can operate for over 30 years even if it performs a checkpoint every 1ms.

3.5.4 Applications

In this subsection, we briefly discuss the implications of the evaluation results on the two major applications of non-volatile computing mentioned in Section 3.2.

Computation across Power Failures

In order to ensure that its state will be properly copied into non-volatile elements, a NV microcontroller must ensure that it has enough energy to finish the `NV erase` operation and perform a new `NV store` operation before it erases its previous checkpoint with an `NV erase` operation. Our experiments indicate that finishing the `NV erase` operation takes 12 pJ and the entire `NV store` operation consumes 172 pJ. Therefore, a microcontroller needs an energy storage for at least 184 pJ. Based on this analysis, a capacitor of 280 pF will be sufficient to ensure that the microcontroller's architecture state will be checkpointed in a non-volatile fashion even on an unexpected power interruption.

Idle Power Reduction

The proposed NV architecture can potentially reduce the idle power consumption by completely turning off a chip after taking an NV checkpoint. In order for this approach to be more energy efficient than simply keeping the microcontroller on, the static energy consumption during the idle period must be more than the energy costs of a power-down.

Design Compiler estimates the static power for our microcontroller to be

493 nW with the dynamic power of 118 uW at 10 MHz. As a comparison, TI MSP430F has the static power of 1.5 uW and the dynamic power of 352 uW. The HSIM simulation shows turning off and on our prototype consumes 35 nJ. Given that our experiments indicate that the non-volatile operations take 344 pJ, the results indicate that our microcontroller can save energy by turning itself off if an idle time is over 72 ms (35nJ/493nW), enabling very fine-grained power gating.

3.6 Related Work

Hybrid Non-Volatile Memory Cell. Researchers have investigated integrating various types of non-volatile memory technologies into either SRAM or flip-flop cell to create non-volatile SRAM and flip-flops [51, 44, 39, 32]. In a high-level, these NV SRAM and flip-flop designs and our hybrid cells share the same approach to integrate non-volatile devices in volatile memory at a cell granularity. However, previous NV cells rely on two-terminal devices such as RRAM, PCRAM, MRAM, and FeRAM while our design uses floating-gate transistors, which have a key advantage in terms of the power consumption in the context of non-volatile computing. Also, the previous cell designs have not been integrated into a system.

Self-Checkpointing Microprocessors. Kothari and Carter proposed to use magnetoelectronic devices to create a non-volatile processor [23]. Their work shares a similar goal with ours, which is to continue an execution over a power interruption. However, the approaches are significantly different. Their architecture relies on off-chip non-volatile memory to provide a back-up instead of

directly checkpointing all on-chip state. Also, because the architecture is based on a two-terminal device that consumes a noticeable amount of current for programming, the architecture takes a periodic snapshot rather than creating an NV snapshot on a power failure.

Non-volatile Microprocessors. A work by Rohm Inc. engineers and Tsinghua University demonstrated in silicon a non-volatile processor augmented in a similar fashion to our work here [52]. They built a per-cell integrated hybrid memory using SRAM and ferroelectric memory. After fabrication and measurement, their reported numbers are quite similar to our simulated ones. This serves as an independent confirmation of the promise of per-cell hybrid memories for non-volatile computing. A more recent work explores the requirements necessary for non-volatile processors to make forward progress using a simulation model based on experiments from a fabricated non-volatile processor [28].

Perpetual Embedded Devices. To enhance capabilities of energy constrained systems, researchers have investigated building extremely low-power platforms [8], having a large energy storage, and harvesting energy from multiple sources [15]. These approaches can significantly alleviate energy constraints by enabling systems to perform more operations and keep their state longer on a given energy budget or providing more energy. However, these systems are still fundamentally volatile. They cannot tolerate an extended power outage or completely eliminate static power consumption. The proposed non-volatile architecture can complement existing techniques to further enhance the platform capabilities under energy constraints.

3.7 Conclusion

In this chapter, we describe a non-volatile architecture and hybrid state elements that enable almost instantaneous checkpointing of the processor state. This capability allows a long computation to reliably execute across even unexpected power failures and also reduces the idle power consumption. Unlike previous approaches that integrate non-volatile memory as discrete components, our approach puts floating-gate transistors into each memory cell. Detailed transistor-level simulations demonstrate that the proposed non-volatile architecture has minimal impacts on regular computations both in terms of performance and energy. The simulation results also suggest that non-volatile checkpoint and restore operations are quite efficient, taking less than 10μs and consuming only 172 pJ.

This project showed great advantage in using a per-cell integrated hybrid memory instead of a traditional architecture, with separate memories located on different substrates.

CHAPTER 4

HYBRID SRAM-DRAM FOR MULTI-THREADED REGISTER FILES

4.1 Introduction

In this part of the thesis, we discuss how replacing a traditionally monolithic SRAM register file with a hybrid memory composed of SRAM and DRAM can significantly reduce energy consumption. Using the density of DRAM allows a smaller total physical area compared to the monolithic SRAM file, while using SRAM allows for the speed a register file needs to function properly.

We targeted this hybrid memory for fine-grained multithreaded-architectures. Fine-grained multi-threading has become a popular architecture style to handle long latency operations such as memory accesses without complex dynamic scheduling hardware. For example, today's Graphics Processing Units (GPUs) use a highly multi-threaded architecture where thousands of threads map to each shader processor [19, 18]. Similarly, SUN's T1 and T2 processors contain simple multi-threaded cores to achieve high throughput with low power consumption [22, 34].

To hide the long latency of memory accesses, a fine-grained multi-threading architecture needs to able to quickly switch among a large number of independent threads. For example, an NVIDIA Fermi stream multiprocessor is reported to support 1,536 simultaneously active threads [18]. As a result, register files on such a multi-threaded architecture are much larger than those in traditional processors. The total register files are reported to be 2 MB in an NVIDIA Fermi GPU [18] and 6 MB in AMD Cayman [20].

In this chapter, we describe a hybrid memory array that tightly integrates embedded DRAM into SRAM cells and study its use in GPU register files as a main application. In our hybrid memory, each SRAM cell is augmented with N DRAM branches ($2N$ 1T1C DRAM cells) in a way that a value can be locally copied between the SRAM cell and one of the DRAM branches within a cell. The memory allows external access to the SRAM cell, but not directly to DRAM branches. The set of data stored in each DRAM branche is called a 'context'. A register file with N contexts can be implemented efficiently with this hybrid array by storing one active context in the SRAM cell and $N - 1$ others in DRAM branches. Registers in the active context can be accessed externally. To access others, a dormant context must be explicitly made "active" by copying from DRAM to SRAM. We call this organization a multi-context register file.

Our study through physical layout and SPICE simulations show that the hybrid cell is far more efficient both in terms of area and energy consumption compared to an SRAM array with N times more cells. For example, the layout indicates that for the same amount of bits stored, a hybrid memory array with 4 DRAM contexts per SRAM cell occupies only 62% of the silicon area compared to an SRAM only array because a DRAM branch is much smaller than a 6-T SRAM cell. Similarly, thanks to the reduced capacitance in word-lines and bit-lines, the hybrid memory consumes much less energy for read and write operations. Also, because accesses from a processing unit only see fast SRAM cells, the hybrid design hides delay of traditional DRAM.

However, the use of the hybrid register file presents new architectural challenges due to the restriction that only a subset of registers can be accessed at a time. First, there may be a context mismatch between two pipeline stages. As

an example, an instruction decode stage may need to read from context 1 while a write-back stage needs to write to context 2. Unfortunately, reading from and writing to two different contexts are not supported by our multi-context hybrid register file. Second, the use of DRAM structure implies that the state is only kept for a short period of time (a few milliseconds) and may need to be refreshed periodically. Finally, context switches may introduce new pipeline stalls that do not exist for today's register files.

To address these architectural challenges, we show how the GPU pipeline and scheduler can be changed to efficiently handle the above challenges associated with hybrid multi-context register files. A careful investigation shows that a small buffer per context can solve the problem of a context mismatch between the decode and write-back stages by delaying write-backs until the corresponding context becomes active again. The worst-case size of the buffers can be bounded based on the pipeline depth. Interestingly, the DRAM refresh turns out to be almost free in fine-grained multi-threading architectures because each context is likely to be used by a processing unit within a refresh period anyways even without explicit refresh operations. Finally, our architecture hides a latency to switch a context by banking a register file and modifying the scheduler to preload a context in the background.

We studied the implication of the hybrid multi-context register file on the overall GPU performance and the register file energy consumption, using a cycle-level GPU simulator. Fortunately, we found that the performance degradation is minimal with 2 or 4 context register files. For example, with a 3 cycle context switching latency, the average performance degradation is only 0.5% for 2-context hybrid memory and 1.4% for 4-context memory. On the other hand,

the hybrid memory provides significant area and energy savings. The area and energy estimates derived from layout, SPICE simulations, and architecture simulations indicate that the 4-context hybrid register file can be implemented with 38% less area and consume 68% less energy on average compared to the traditional SRAM-based register file.

We make two main contributions; one in the area of memory cell designs and one in GPU micro-architecture. While embedded DRAM has been used in many different contexts including L3 caches [54], the DRAM array has mostly been designed as a standalone structure. We introduce a novel way to combine DRAM branches into each SRAM cell so that a context switch between them can be performed in a parallel fashion with low overheads (Chapter 4.2). In addition to the memory design, we propose a set of architectural techniques and a modified scheduler to address limitations of a multi-context register file (Chapter 4.3). These architecture techniques are applicable not only to the proposed SRAM-DRAM hybrid memory, but to other register file implementations that limit access to its content.

The rest of the chapter is organized as follows. Chapter 4.2 introduces our SRAM-DRAM hybrid memory design and explains how the memory cell operates. Using the hybrid memory, Chapter 4.3 discusses how a typical GPU pipeline and a scheduler can be modified to incorporate the register file based on the hybrid memory. Chapter 4.4 evaluates both the hybrid memory and the overall GPU architecture through circuit-level and architecture-level simulation studies. We discuss related work in Chapter 4.5. Finally, Chapter 4.6 concludes.

4.2 Multi-Context Memory Using SRAM-DRAM Hybrid Cells

In a traditional memory array, any memory location may be accessed at any given time, often with the same cost. Such a uniform organization often implies that all locations are implemented with a single memory technology. Here, we introduce a notion of a multi-context memory array and show how SRAM and DRAM technologies can be tightly integrated to provide an efficient implementation.

4.2.1 Multi-Context Memory

We define a multi-context memory as a memory array where cells are partitioned into multiple contexts, out of which only one active context is accessible at a time. To access other contexts, termed dormant contexts, the active context must be explicitly switched. Conceptually, an N-context memory consists of a memory array for one active context and N back-up arrays for dormant contexts.

For accesses to the active context, the multi-context memory behaves identically to a traditional SRAM array. Normal read and write operations are not affected by having multiple contexts except that they can only access the active context. To manage data in the dormant contexts, two new operations to store and load contexts are introduced:

- *Store context i*: Save the active context to a back-up array i. The active context is not affected and remains available.

- *Load context i*: Copy a context from a back-up array *i* to the memory array for the active context. This operation overwrites data in the active context.

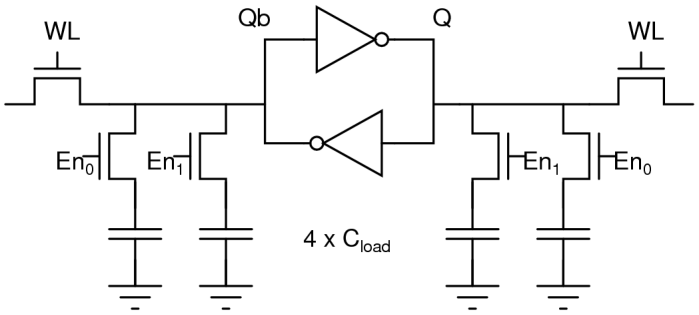
While performing a store or load of a context, the multi-context memory will not be able to service regular read and write operations. The regular read or write will be delayed until the store or load of a context is complete.

4.2.2 SRAM-DRAM Hybrid Cell

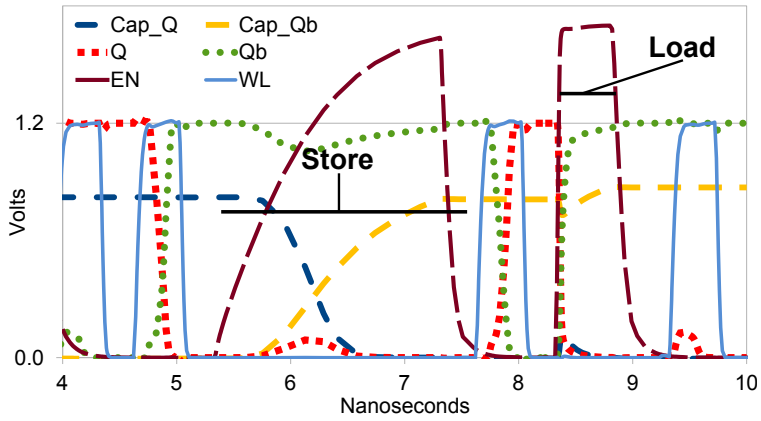
The multi-context memory, thanks to its explicit management of active and dormant contexts, allows a memory array to potentially be implemented more efficiently as a hybrid of multiple memory technologies. We propose to tightly integrate SRAM and DRAM technologies to construct a new hybrid memory cell that is suitable for a multi-context memory array.

To build a N-context SRAM-DRAM hybrid, a typical 6-transistor SRAM cell is augmented with N 1-transistor 1-capacitor DRAM cells at both `Q` and `Qb` nodes. Figure 4.1(a) shows the circuit design for $N = 2$. More pairs of DRAM cells can be attached to support more dormant contexts. The DRAM capacitors are isolated from the SRAM's cross-coupled inverters by DRAM enable transistors during normal read/write operations. By controlling the enable transistors, a value can be copied between the SRAM cell and DRAM cells. The circuit uses a pair of DRAM cells to store one dormant context.

We choose an SRAM cell to store the active context as it provides fast access. Dormant contexts are stored in DRAM cells in order to minimize the area and the power consumption, providing extra capacity at minimal cost. As an exam-



(a) Hybrid memory cell circuit



(b) Storing and restoring a context

Figure 4.1: A hybrid SRAM-DRAM cell with an SRAM bit for the active context and DRAM bits for dormant contexts.

ple, the area of an embedded 1-T DRAM cell is reported to be as small as 15% of an SRAM cell in a 65nm process technology [5].

The cell in the figure is capable of storing 2 contexts (1 bit per context) using the two DRAM branches (4 DRAM cells). The SRAM cell is only used to make a context in DRAM available and cannot be used as extra storage because a read from DRAM overwrites active context in the SRAM cell. As a result, an active context in SRAM must be copied to a DRAM before another context is restored.

Read and write operations for the active context in the hybrid memory cell are conducted in the same manner as with a traditional SRAM cell as the active context is stored in the SRAM cell. During read and write operations, the DRAM enable transistors connecting the SRAM cell and the DRAM capacitors are turned off, isolating the DRAM cells from the SRAM.

A context store operation is performed by writing a bit in the SRAM cell into the selected dormant DRAM context as shown in Figure 4.1(b). The values on the Q and Qb nodes are copied into a pair of DRAM capacitors, one connected to Q and the other connected to Qb. Loading a dormant context into the SRAM is done by reading from the selected pair of DRAM capacitors. The data in the SRAM context would be lost unless first saved.

Both context store and load operations are performed by turning on the DRAM enable transistors, connecting Q and Qb nodes to the corresponding DRAM capacitors. At a circuit level, we design the DRAM capacitance to be larger than the Q and Qb node capacitance, and the load and store operations are differentiated by the ramp time of the enable signal for transistors connecting DRAM capacitors to SRAM nodes. A fast ramp time (< 0.1 ns in our circuit) connects the larger DRAM capacitance to the SRAM state node abruptly and the SRAM node swings toward the DRAM capacitance (load a context from DRAM). A slower ramp time allows the SRAM state node enough time to drain or fill the DRAM capacitor to its own value (store a context to DRAM). Wordlines and bitlines stay off. The waveforms in Figure 4.1(b) depict storing a 0 to the DRAM, writing a 1 to the SRAM, then loading the stored 0 from the DRAM.

The functionality of the hybrid cell has been studied via HSPICE simulations and a layout of a small array in IBM 65nm process technology. The SPICE

simulations validated the correctness of the operations. The retention time of a DRAM content is mainly determined by the size of the DRAM capacitor and the length of the DRAM enable transistor, which affects the leakage current. In our circuit design, simulations suggest that a retention time of around 2.3 ms can be achieved with a capacitor size of 7.5 fF. We also studied the effects of additional capacitance, the timing of enable signals, the impact of process variations, etc., to further verify the hybrid memory functions. Section 4.4.2 provides detailed evaluations of the area, energy consumption, and delay of the hybrid memory array compared to traditional SRAM arrays.

4.2.3 Strengths and Weaknesses

Utilizing the efficiency of DRAM, the SRAM-DRAM hybrid memory can provide the same storage capacity with significantly less silicon area and energy consumption, compared to a traditional SRAM array. One SRAM-DRAM hybrid cell with 2 DRAM contexts can store the same amount of data as two SRAM cells. The layouts suggest that a 2-context hybrid memory cell occupies 1.8 times the area of a single SRAM cell and a 4-context hybrid cell consumes only 2.4 times the area of a single SRAM cell. As a result, the 4-context cell reduces the area almost by half compared to 4 SRAM cells. The area reduction leads to lower leakage power and also lowers dynamic power consumption for read and write operations because bitlines and wordlines for the memory array become shorter.

In exchange for the efficiency, the hybrid memory trades-off data accessibility. The hybrid memory does not allow accesses to data in a dormant context

unless there is an explicit context switch. On the other hand, in a traditional SRAM array, any location can be accessed without any restriction. The context switches cost additional latencies and energy consumption, especially if contexts need to change frequently. The use of DRAM cells also imply that explicit refresh operations are required in order to keep state in dormant contexts for an extended period because the charges in the DRAM capacitors will eventually leak out.

4.3 GPU Architecture with Multi-Context Register File

In order to effectively apply the hybrid memory as register files in fine-grained multi-threading architectures such as GPUs, the architecture needs to be able to properly compensate or hide the weaknesses of the hybrid memory, namely limited accessibility and overheads for context switches and DRAM refreshes. In this section, we discuss how a GPU pipeline and a thread scheduler can be changed to utilize the hybrid memory with minimal impact on performance.

4.3.1 Baseline GPU Pipeline

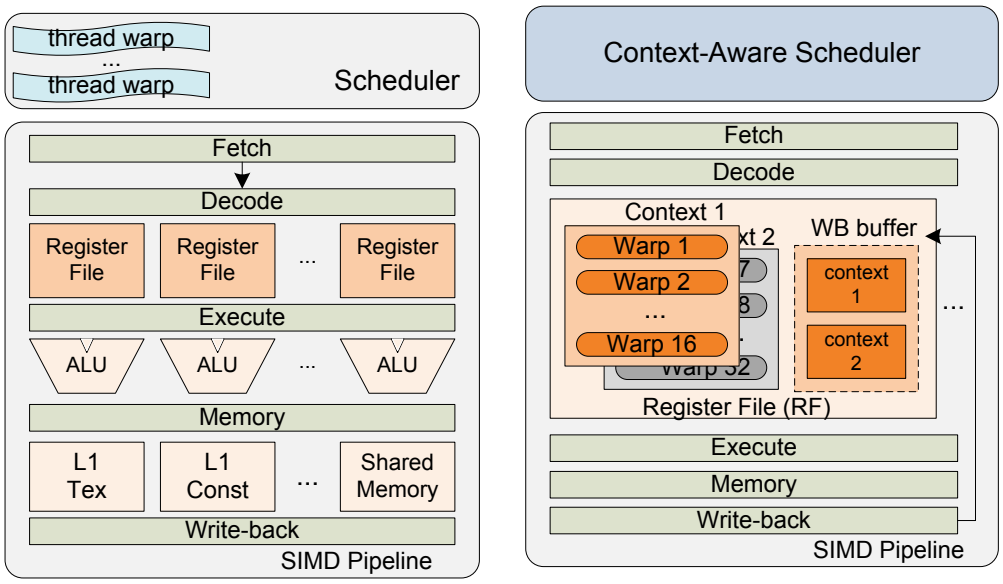
We use a typical General-Purpose GPU (GPGPU) processing pipeline as the baseline for our discussions. Computation in a GPGPU occurs in a highly parallel manner. A GPGPU consists of many cores, often called stream multiprocessors (SM), each of which again contains multiple execution units. The GPGPU pipeline often issues threads in groups of 32, called a warp. Warps are issued atomically and all threads in a warp must be ready to execute in order for a

warp to be issued. Threads in a warp get executed in parallel by multiple execution units in a core. As an example, NVIDIA GT200 with 8 execution units takes 4 pipeline cycles to execute a warp with 32 threads [19].

To fully utilize the pipeline and hide a long memory access latency, each GPGPU core supports thousands of threads. A hardware scheduler operating on warp granularity determines which warps are ready and selects one to issue. Typically, there is no penalty in switching from one warp to another. Therefore, even if some warps are stalled for long latency operations, the overall pipeline can be kept busy with other warps.

For this fine-grained multi-threading scheme, each thread needs its own set of architectural registers in the core register file. As a result, the register file in a GPGPU core is large. Recent NVIDIA GPGPU register files have 32,768 entries of 32 bits each, resulting in 128KB per core and 2MB for a chip [18]. As a comparison, typical high-performance general purpose processors have much smaller register files in the range of 8 to 16KB per core as they contain low hundreds of entries of 64 bits each.

Figure 4.2(a) shows the baseline GPU pipeline that uses a typical SRAM-based register file. While commercial GPUs contain specialized blocks for graphics, we only discuss a general processing pipeline because we are concerned with a GPU as a general-purpose platform. Our baseline pipeline consists of fetch, decode, execute, memory, and write-back stages that are similar to the traditional processor pipeline. This view of the GPGPU pipeline has been borrowed from the previous work on GPU modeling [4] based on NVIDIA documents [35]. Register file accesses occur in the decode and write-back stages.



(a) Baseline (b) Multi-context RF

Figure 4.2: GPU processing engine block diagrams with single-context and multi-context register files.

GPGPU cores typically contain multiple execution units that operate in parallel. In this sense, as shown in the figure, the GPGPU pipeline has a collection of multiple parallel pipelines, one for each execution unit. In the following discussions, we will refer to each execution pipeline as a *lane*. The multiple lanes in a core imply that the GPU register file needs to support many reads and writes in each cycle. For example, the NVIDIA GT200 GPU has 8 lanes per core where each instruction may read up to 3 registers and write one register. This implies up to 24 reads and 8 writes per cycle. Because highly-ported register files are expensive, a typical GPU relies on banked register files where each lane reads from its own register file bank. This banked design implicitly restricts each thread to only be able to execute on a particular lane. For example, the 1,024 threads in a GT200 core are statically partitioned into 8 lanes. As a result, one register file bank only needs to contain registers for 128 threads.

4.3.2 Pipeline with Multi-Context Register Files

Figure 4.2(b) shows changes to the baseline GPU pipeline when a multi-context memory register file is used. As in the baseline case, the register file is banked so that each bank only needs to support one lane. However, each bank will use hybrid memory cells to reduce its area and power overheads. As shown in the figure, with a multi-context register file, only a subset of the registers in an active context are accessible from the pipeline. Registers in the other contexts are stored in dormant arrays and need to be explicitly switched into the active context to be accessible. For example, a 16-KB SRAM bank for a traditional register file can be replaced with a 4-context memory array with 4-KB contexts (4-KB SRAM backed by 4 4-KB DRAM contexts) or a 2-context array with 8-KB contexts (8-KB SRAM baked by 2 8-KB DRAM contexts), etc.

In multi-context register files, the registers must be divided into each context. In a GPGPU, a logical granularity for this assignment would be a warp, as threads from a warp are scheduled together, implying that their registers will be accessed together. To minimize the number of unnecessary context switches in the register file, the context assignment should keep threads from a warp in the same context and minimize the number of overall contexts. We achieve this objective by sequentially assigning each warp into each context in order. For example, if only the first 16 out of 32 warps are active, and there are four contexts for a register file, warp 1 to 8 will be assigned to the first context, and warps 9 to 16 will be assigned to the second context, using only 2 of the 4 contexts.

While a multi-context register file can bring significant advantages in area and power consumption, its limitations also pose new challenges in the architecture design. The following discussions address how the three major challenges,

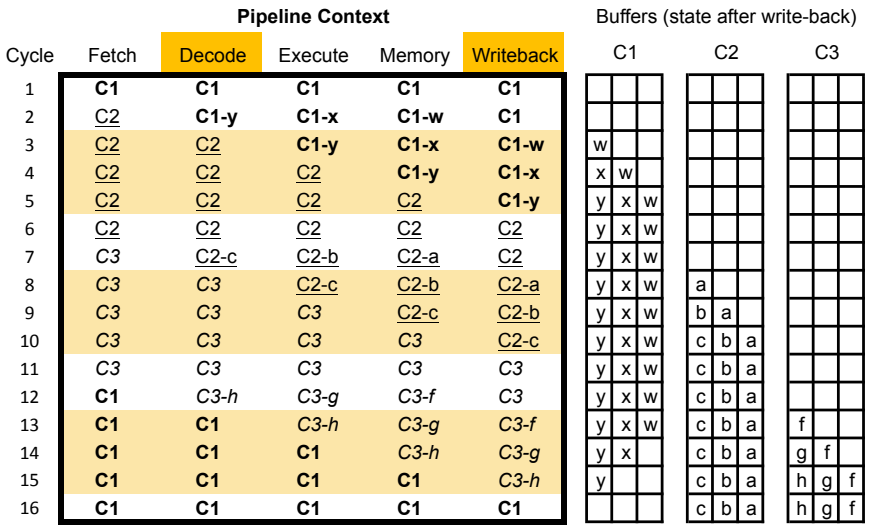


Figure 4.3: Context mismatch between pipeline stages and our solution with context write-back buffers.

namely context mismatches among pipeline stages, context switch overheads, and DRAM refresh overheads, can be addressed in our architecture.

Context Mismatch between Pipeline Stages

The multi-context register file implies that a processing core may switch an active context depending on which warp is being issued. Therefore, different stages of the pipeline may need data from different contexts simultaneously. For example, a register in context 1 may be used in the write-back stage, while a register in context 2 needs to be read as an operand in the decode stage. The accesses to the two different contexts pose a conflict because the multi-context register file can only allow accesses to one context at a time. Figure 4.3 illustrates an example of such a conflict with three contexts: C1, C2, and C3. In cycle 3, C2 needs to read registers while C1 needs to write a result back.

A naive way to avoid such a structural hazard is to complete all pipeline stages for instructions from one context before switching register contexts and issuing instructions from a new context. Essentially, this method stalls the decode stage until preceding instructions complete on every context switch. As a result, this approach can add a large number of stalls and significantly degrade performance.

To avoid stalls, a warp from a new context must be able to continue its execution without waiting for processing of warps from the previous context to be completed. Unfortunately, this optimization implies that the write-back stage of the pipeline cannot write results back because the register file is already switched to another context. To address this problem, we add a small write-back buffer for each context to the register file. On a context switch, a write-back stage can put its result into the buffer for its context while the decode stage can read from the new context. Later, the entries from the write-back buffer can be put into the register file when the corresponding context becomes active again.

Figure 4.3 shows operations of the multi-context register file with per-context write-back buffers shown on the right side. After the first context switch from C1 to C2 in cycle 3, results from the instructions belonging to C1, tagged with w,x,y, are written to C1's write buffer. When the context is switched back to C1 in cycle 13, actual C1 write-backs to the register file take place while C3 writes to its buffer. Note that there will be no register port contention for such delayed write-backs. After a context switch, the write-back stage in the pipeline will write to the buffer for the previous context, while the write-back buffer for the current context empties out to the register file. In effect, this scheme delays a particular context's write-back to the next time it is switched into the active

context. The size of each write-back buffer can be bounded by the number of pipeline stages between the decode and the write-back.

Note that on a read, with this optimization, the register file now needs to check the current context's write-back buffer. If there is a match, the value from the write-back buffer needs to be used instead in the pipeline. Because the write-back buffers are small, with only several entries to tentatively delay write-backs, a look-up to the write-back buffer can be performed in parallel to a register file read without any performance impact.

Context Switch Latencies

With the multi-context register file, the pipeline can only access registers from an active context. Once all warps in the active context become blocked, the core needs to explicitly make one of the dormant contexts active before it can issue warps from other contexts. Unfortunately, these register context switches can stall the decode stage of the pipeline because the register file becomes inaccessible during the switch. To alleviate the context switch overheads, as discussed before, we carefully place warps into contexts in a way that minimizes the number of active contexts given the number of active warps. This warp placement aims to reduce the number of context switches. However, because context switches may still happen rather frequently, we also need a way to hide the context switching latency without stalling the pipeline.

To hide the context switch latency, we propose to use two sub-banks for a register file bank in each lane so that a context switch of one sub-bank can be performed in the background while the other sub-bank is being used by

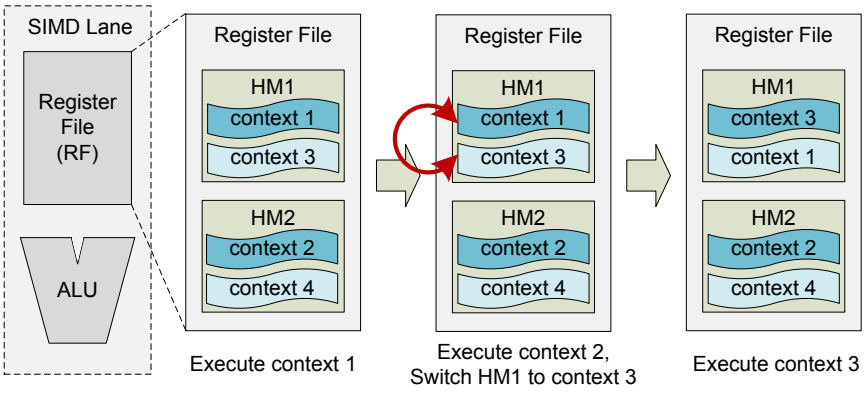


Figure 4.4: A background context switch using two register sub-banks per lane.

the pipeline. Figure 4.4 illustrates how banking enables background context switches. As shown in the figure, one hybrid memory bank is split into two sub-banks, each with half the size of the original one. For example, a 16-KB 2-context memory array (8-KB SRAM with 2 8-KB DRAM) is replaced with two 8-KB 2-context arrays (4-KB SRAM with 2 4-KB DRAM). In the following discussion, we call the two sub-banks as Hybrid Memory 1 (HM1) and Hybrid Memory 2 (HM2). HM1 contains context 1 and 3, and HM2 contains context 2 and 4.

The pipeline starts issuing warps from an active context of one of the sub-banks, say context 1 from HM1. When HM1 needs to switch its context, the pipeline can issue warps from the active context of the other sub-bank, say context 2 from HM2. While executing warps from HM2, HM1 can switch from context 1 to context 3 in the background. By the time that HM2 needs to switch its context, the context switch for HM1 would be completed so that warps can be issued from HM1 without stalling the pipeline. If warps from HM2 are blocked before the context switch is done, the switching latency is only partially hidden. Section 4.3.3 discusses the warp scheduling algorithm, which controls context

switches, in more detail.

DRAM Refresh

Our hybrid memory cell uses DRAM capacitors as storage elements for dormant contexts. Because DRAM leaks charge over time, this construction implies that dormant contexts need to be periodically refreshed - loaded to the active context and stored back. For example, our reference design has a retention time of 2.3 ms. A longer time could be achieved by increasing the DRAM capacitor size, at the cost of more energy expended during store operations.

To ensure that each dormant context is refreshed within the retention time, we add a refresh timer for each context to keep track of the number of cycles since the last time the context has been written to DRAM. If a refresh timer is close to the retention limit, the schedule is forced to bring the corresponding context to SRAM as an active context. Obviously, this operation also requires the current active state to be copied into DRAM. This refresh operation may incur pipeline stalls if there is no warp to issue from another sub-bank.

In practice, however, our experiments indicate that fine-grained multi-threading architectures tend to re-schedule each warp within a reasonably short period. As a result, each dormant context almost always gets brought back to SRAM as an active context before its refresh timer reaches a retention limit. As a result, we found that the DRAM refresh typically does not interrupt normal pipeline operations.

4.3.3 Context-Aware Warp Scheduling

In our architecture, a scheduler needs to perform two major functions. First, similar to the baseline GPU, the scheduler needs to select which warp will be issued in each cycle. However, unlike the baseline, the scheduler is restricted to choose a warp from the active contexts of two register file sub-banks. Additionally, the scheduler needs to also manage multiple contexts for each sub-bank by deciding when a context switch should happen and which context should become active. The context switches need to be carefully managed considering multiple factors including DRAM refreshes, context switch overheads, and fairness.

In this subsection we discuss how the warp scheduling algorithm can be augmented to accommodate additional constraints from the hybrid memory. More specifically, there are three major design considerations for the scheduler:

- *Correctness*: The scheduler must ensure that dormant contexts in DRAM are preserved by reloading each context before it reaches the DRAM retention limit.
- *Efficiency*: To minimize pipeline stalls and context switch energy, the scheduler needs to minimize unnecessary context switches and intelligently schedule context switches to hide their latencies.
- *Fairness*: The scheduler needs to be fair to all warps, providing equal opportunities to execute. Unfair scheduling can lead to a noticeable performance degradation by leaving only a small number of warps towards the end of program execution.

While it is relatively straightforward to deal with the DRAM refresh requirement, we found that there is a fundamental tension between reducing the number of context switches and improving fairness. In order to minimize context switches, the scheduler should issue warps from one active context as long as there is a "ready warp" (warp that is ready to be issued) in the context. However, such a greedy algorithm may lead to unfairness, especially when a subset of contexts have more active warps than others; contexts with more active warps are more likely to get chances to execute. We found that the unfairness can significantly degrade performance for some applications by leaving a small number of warps to run towards the end of an execution without enough warps to fully utilize the pipeline.

Our scheduling algorithm deals with the tension between context switch overheads and fairness by setting a threshold on how long that each context can run without switching to another context with ready warps. While this threshold does not strictly guarantee fairness, we found that the scheme works quite well in practice, close to more complex scheduling algorithms with stronger fairness guarantees. The evaluation section provides further discussions on the trade-off between context switches and fairness based on simulations.

Figure 4.5 shows the flowchart for the new scheduler. To illustrate how a scheduling algorithm is changed for the multi-context register files, we use a simple baseline scheduling algorithm, which selects a ready warp in a round-robin fashion. To aid its decisions, the scheduler maintains three types of counters per context: refresh counter (*rcnt*), schedule counter (*scnt*), and ready-warp counter (*wcnt*). The refresh counter keeps track of when a DRAM context needs to be refreshed. The schedule counter indicates how many times a warp in a

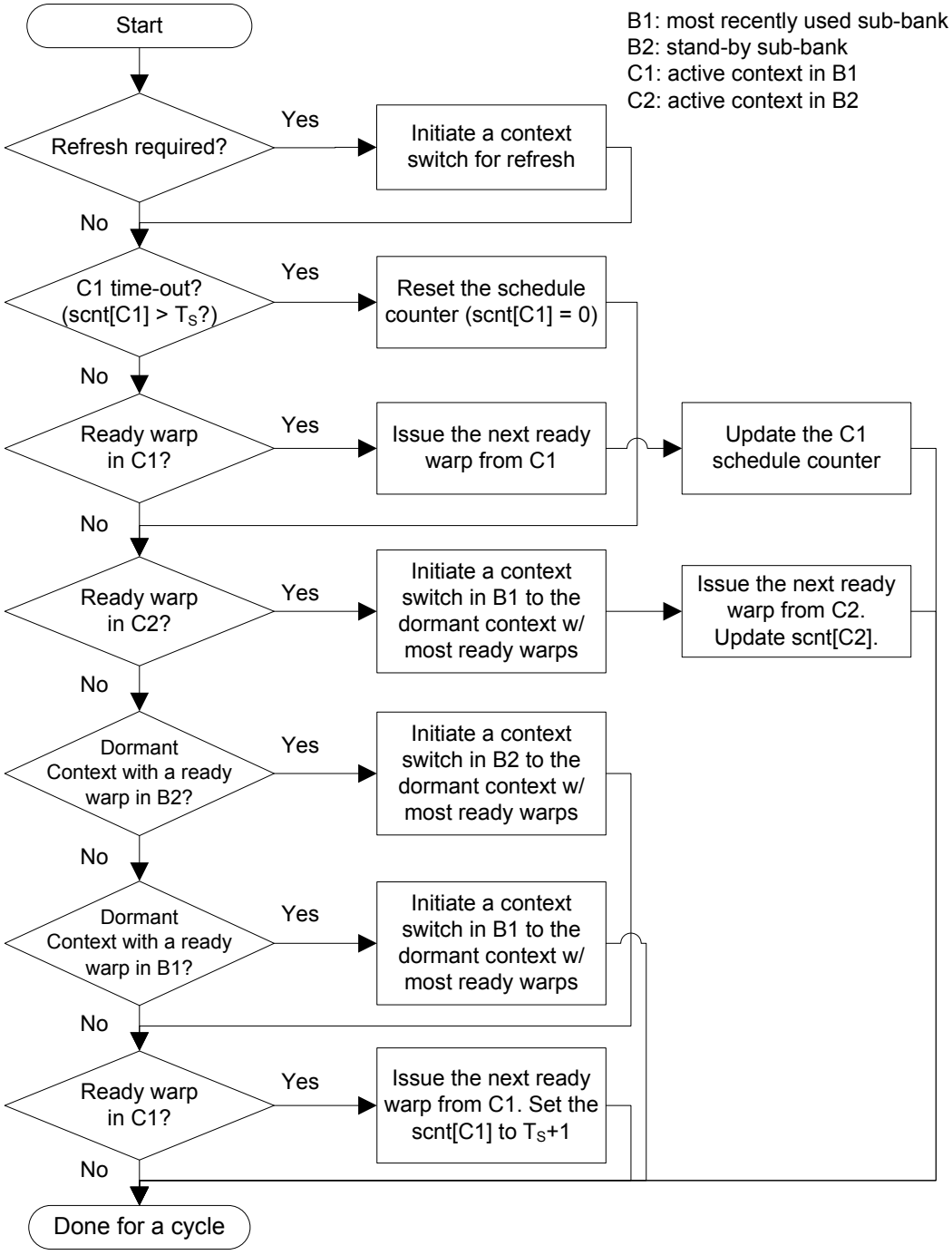


Figure 4.5: Flow chart for warp scheduling and register file context switching.

context has been checked for possible issue by the round-robin scheduler without a forced context switch. The ready-warp counter indicates how many warps are ready for each context.

As shown in the figure, in each cycle, the scheduler first checks if any dormant context needs to be refreshed, and initiates a context switch to make the dormant context active if necessary. The corresponding sub-bank is excluded from scheduling decisions until the switch is completed.

To determine which warp to issue in a given cycle, the scheduler first tries to select a ready warp from the active context (C1) of the most recently used sub-bank (B1) in a round-robin fashion. If the schedule counter for C1 ($scnt[C1]$) reaches a fairness threshold (T_S) or there is no ready warp in C1, the scheduler tries to pick a ready warp from the active context (C2) of the other sub-bank (B2, called stand-by sub-bank) and initiates a background context switch in B1 if successful. If there is no ready warp in the stand-by sub-bank (B2), the scheduler searches for a dormant context with a ready warp and initiates a context switch. To reduce future context switches, the scheduler selects a dormant context with the largest number of ready warps on a context switch. Finally, if no warp from another context is ready, a ready warp from the current active context (C1) is allowed to issue even if the threshold is reached.

While not shown in the flowchart to keep the chart simple, our scheduler implementation contains one more optimization to hide context switch latencies. In each cycle, while issuing a warp from one sub-bank, the scheduler checks how many ready warps are left in the current active context of that sub-bank. If the number is low, just enough to hide a context switch latency, and there is no ready warp in the active context of the stand-by sub-bank, the scheduler

initiates a context switch in the stand-by sub-bank.

4.4 Evaluation

This section evaluates the performance, area, and energy consumption of the proposed hybrid memory and GPU architecture through detailed simulations.

4.4.1 Experimental Methodology

To validate functionality and study performance of the hybrid memory, we used SPICE simulations along with cell layout. The SRAM-DRAM hybrid cell was simulated in HSPICE using IBM 65nm process technology transistor models. The cell was laid out using custom layout with IBM 65nm design rules. For the embedded DRAM, the area and placement are estimated based on the previously reported numbers from an eDRAM array [5]. The hybrid memory arrays were laid out in sizes from 16x1 to 16x64, extracted and simulated to obtain area and energy measurements. Array area and energies for larger arrays are estimated by scaling the bitline and wordline capacitances, which are predictable from the array size. We also checked this scaling method by designing several memories with a commercial memory compiler.

In this study, we use single-port SRAM arrays rather than multi-ported ones to evaluate our hybrid memory design. We believe that GPU register files are often heavily banked and use single-port SRAM arrays to save silicon area. The single-port configuration also provides conservative estimates for hybrid memory overheads. In multi-ported arrays, a part of the DRAM branch overhead

Parameter	*Value*
Shader cores	30
Threads per core	**1024**, 2048
Threads per warp	32
Warps per core	**32**, 64
Registers per core	**16384**, 32768
Register file size per core	**64KB**, 128KB
Register file banks per core	32
Register file bank size	**2KB**, 4KB
Register file sub-banks	2
Execution units per core	16, **32**
Hybrid Memory Register File	
Contexts per cell	2, 4, 8
Context switch latency	**3**, 4, 5
Scheduler counter threshold	**10,000**

Table 4.1: GPGPU-Sim parameters and hybrid memory configurations. Bold denotes default values.

can likely be hidden by additional bit-lines.

For GPGPU performance estimates and counting events for energy estimates, we modified GPGPU-Sim v2.1.1.b [4] to model the new scheduler and the hybrid register file. Simulator parameters are shown in Table 4.1. In the default configuration, each shader core has 32 execution units and includes a 64-KB register file. As a result, the core can complete one warp per cycle. The register file is assumed to be split into 32 banks of 2KB each, one per execution unit. Each bank is also assumed to have two 1-KB memory arrays (sub-banks), which can be used for background context switches as discussed in Section 4.3.2. For fair comparisons, both the baseline SRAM register files and the hybrid register files use the same sub-bank configuration.

For the hybrid register file, we studied 2, 4, and 8-context configurations to implement each sub-bank. For example, a 1-KB sub-bank array can be imple-

mented with a 1-KB SRAM array, a 2-context hybrid array with 512 bytes (8 warps) per context, a 4-context hybrid array with 256 bytes (4 warps) per context, or 8-context hybrid array with 128 bytes (2 warps) per context. To study the impact of various architecture parameters on the hybrid register file performance, we tried different context switch latencies (3, 4, and 5), a narrower pipeline (16 execution units), and more threads per core (2048 threads with 128-KB register file).

The energy consumption of a register file is estimated by combining results from the circuit and architecture simulations. The read, write, and context switch energies for a 1-KB sub-bank are obtained from SPICE simulations. Then, the overall energy consumption is computed by multiplying the per-event energy with the number of events that are counted by the architecture simulator.

Table 4.2 shows the benchmarks that are used in the simulations and their characteristics. The benchmarks were taken from the GPGPU-Sim benchmark suite [4] and the Rodinia GPGPU benchmark suite [10, 11]. They comprise compute-heavy applications meant to be run in the massively parallel environment of a GPU, and cover a wide range of register file usage. Some benchmarks require a small number of registers while others heavily stress the register file.

4.4.2 Hybrid Memory

For the hybrid SRAM-DRAM memory cell, we studied configurations with 2, 4, and 8 DRAM contexts attached to each SRAM cell. We refer to these designs as 2-context, 4-context, and 8-context configurations.

Name	*Description*	*Warps/ core*	*Inst. count*	*Baseline IPC*
LPS	3D Laplace Solver	16	82M	551
BLK	Black-Scholes option pricing	24	196M	804
NQU	N-Queens Solver	3	1.3M	42.4
MUM	MUMmerGPU	32	75M	62.2
RAY	Ray Tracing	12	65M	722
STO	StoreGPU	4	124M	734
WP	Weather Prediction	6	217M	215
BFS	Breadth First Search	32	17M	48.7
NN	Neural Network Digit Recognition	30	68M	29.2
R_BFS	Breadth First Search	32	484M	97.4
R_HS	Thermal Simulation	16	80M	830
R_LUD	LU Decomposition	32	40M	63.5
R_NW	Needleman-Wunsch DNA alignment	5	218M	48.7

Table 4.2: Benchmark characteristics. Last 4 benchmarks from Rodinia suite. Warps/core: maximum number of warps assigned at a time per core.

Cell-level Discussion

Figure 4.6 shows our layout for the 2-context hybrid SRAM-DRAM cell. The block diagram in the top half of the figure shows the floorplan of an array. The DRAM cell 'leaves' from adjacent SRAM cells would overlap and share space, allowing for a compact design. The image in the bottom half of the figure is a screen shot of the actual layout for the 2-context SRAM-DRAM cell, with the SRAM and DRAM cells highlighted.

Area estimates from extracted layout of an SRAM and hybrid SRAM-DRAM cell appear in Table 4.3. Adding 2 contexts to the SRAM cells costs another 83% area overhead compared to one SRAM cell. With 4 contexts, we note that the

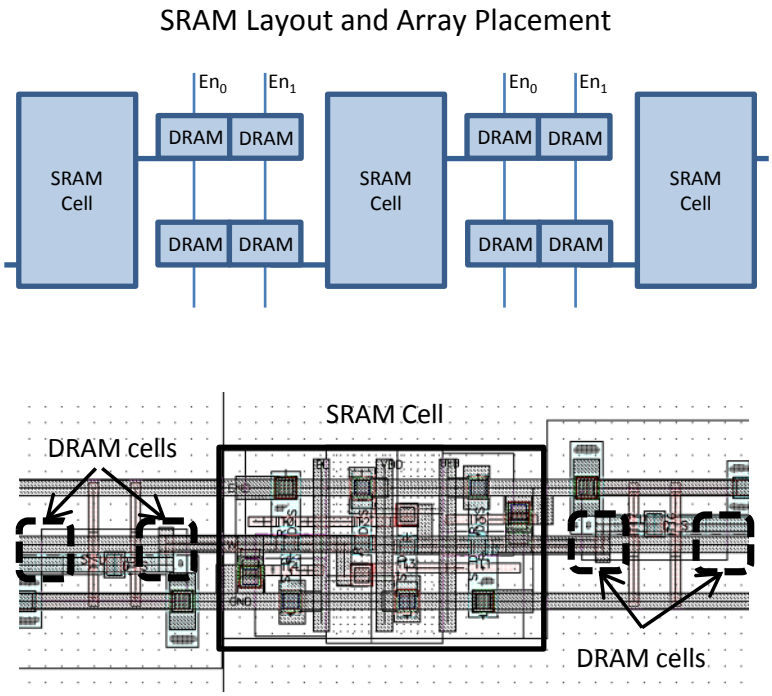


Figure 4.6: SRAM with 2 contexts layout.

Parameter	*SRAM*	*Hybrid 2-context*	*Hybrid 4-context*	*Hybrid 8-context*
Width (μm)	1.84	3.37	4.47	6.67
Height (μm)	0.7	0.7	0.7	0.7
Area (μm^2)	1.29	2.36	3.13	4.67
Relative sizes	100	183.2	242.9	363.5

Table 4.3: SRAM-DRAM hybrid memory cell area comparison.

area overhead is only 140% more than a single SRAM cell. Compared to 4 SRAM cells holding the same amount of data, the area of a hybrid SRAM-DRAM cell with 4 contexts is 39% smaller.

At the cell level, the additional DRAM branches can have noticeable impact on memory latencies. Simulations indicate that the read delay at the cell level is increased by 70% in the 8-context hybrid cell. The write delay increases by 116%. The 4-context hybrid cell read and write delays are increased by 38% and 64%, respectively.

Array-Level Discussion

In memory arrays, the read/write latency and energy are dominated by the bitline and wordline capacitances. As a result, the array-level characteristics are largely determined by the size of an array, and the additional DRAM capacitance at each SRAM cell only has a minor impact. Table 4.4 shows the array-level energy and area estimates for different array sizes. Because the hybrid memory array is physically much smaller and requires fewer SRAM cells than the corresponding SRAM array of equal data storage, its bitline capacitance is also smaller, which results in much lower energy consumption for read and write operations. For example, we find that a 1-KB SRAM array consumes 7.32pJ for a read whereas a 1-KB (2048 cells) 4-context hybrid array only consumes 2.03pJ per read operation. This represents an reduction in read energy of 72%. The hybrid memory also shows a similar reduction in write energy. The table also shows that the energy savings for regular read/write operations are more significant with more DRAM contexts thanks to further reduction in area and memory cells.

Bytes Stored					
-	128	256	512	1024	2048
SRAM array energy cost					
Cells in array	1024	2048	4096	8192	16384
Read energy (fJ)	919	1830	3660	7320	14600
Write energy (fJ)	757	1510	3030	6060	12100
Leakage energy (nW)	202	400	795	1590	3170
Hybrid SRAM-DRAM, 2 context energy cost					
Cells in array	512	1024	2048	4096	8192
Read energy (fJ)	526	1040	2070	4140	8270
Write energy (fJ)	502	944	1830	3600	7140
Context switch (pJ)	5.12	10.2	20.5	40.9	81.9
Leakage energy (nW)	149	296	588	1170	2340
Hybrid SRAM-DRAM, 4 context energy cost					
Cells in array	256	512	1024	2048	4096
Read energy (fJ)	287	535	1030	2030	4020
Write energy (fJ)	227	456	914	1830	3660
Context switch (pJ)	3.69	7.38	14.8	29.5	59.0
Leakage energy (nW)	71.0	142	284	568	1140
Hybrid SRAM-DRAM, 8 context energy cost					
Cells in array	128	256	512	1024	2048
Read energy (fJ)	178	303	553	1050	2050
Write energy (fJ)	116	239	485	977	1960
Context switch (pJ)	2.01	4.03	8.05	16.1	32.2
Leakage energy (nW)	39.4	76.0	149	295	588
Area in μm^2					
SRAM	1529	3059	6118	12235	24471
2 context SRAM-DRAM	1471	2871	5673	11275	22480
4 context SRAM-DRAM	1057	1985	3843	7559	14990
8 context SRAM-DRAM	935	1628	3014	5786	11330

Table 4.4: SRAM-DRAM hybrid memory array energy and area estimates.

At a circuit-level, there is a large design space for a memory array. For example, peripheral circuits can be sized differently in order to provide a different trade-off point between delay and energy. In this study, given that we use the hybrid memory to replace an SRAM-based register file in a pipeline, we sized read and write circuits so that the read/write latency of all SRAM and hybrid memory arrays are roughly identical. For example, a hybrid memory array with additional DRAM capacitance uses larger read/write drivers compared to the SRAM array with the same number of cells. Larger arrays also use larger read/write circuits. The read and write latencies for our arrays are 130ps and 140ps, respectively. We also note that the read and write circuits could be further optimized.

In addition to regular read and write operations, the hybrid SRAM-DRAM memory needs to move data between SRAM and DRAM. Table 4.4 shows the estimated context switch energy for different array sizes, which includes energy consumption within each cell as well as energy to change control lines (EN). For typical array sizes, the estimated energy consumption of a context switch is comparable to a few tens of read operations. In terms of latency, our SPICE simulations show that the hybrid SRAM-DRAM cell can switch a context (one SRAM-to-DRAM store and one DRAM-to-SRAM load) under 4 ns (see Figure 4.1(b)). This is a conservative estimate and we believe that the switch can be even faster if necessary. Given that current GPU cores run at around 700 MHz, a context switch will take about 3 core clock cycles. To see the impact of switching latency, we also simulated 4 and 5 clock cycle latencies in later studies.

Table 4.4 also shows area estimates based on the layout. The estimates only include the area of an actual cell array without any peripheral circuitry. Mea-

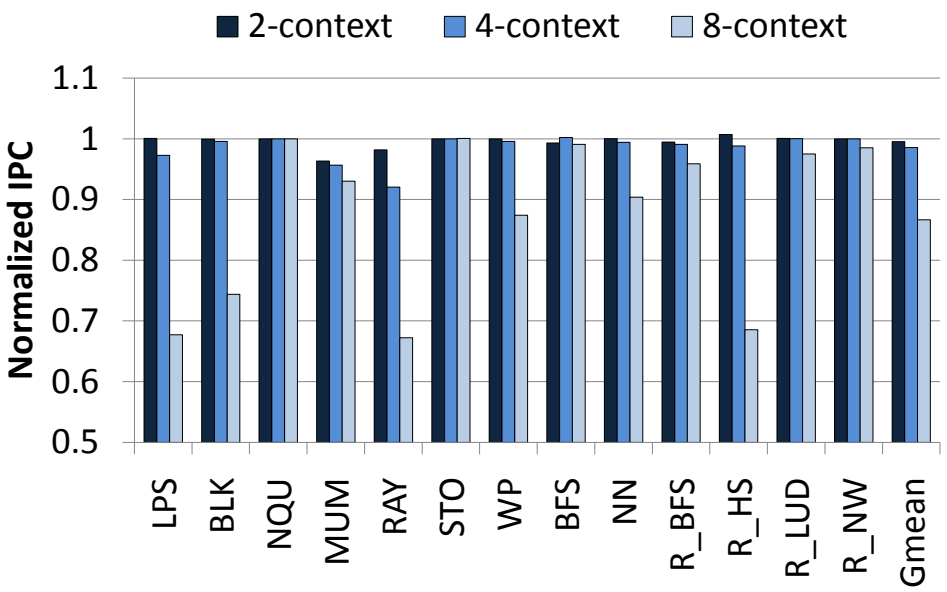


Figure 4.7: Performance comparisons for hybrid register files (2, 4, and 8 contexts) under 3 cycle context switch latency and 32 execution units per core. The performance is normalized to the baseline SRAM register file.

surements are shown in μm^2. The 2-context hybrid memory arrays are roughly 7% smaller than SRAM arrays with the same capacity, while the 4-context and 8-context memories use 38% and 52% less area, respectively.

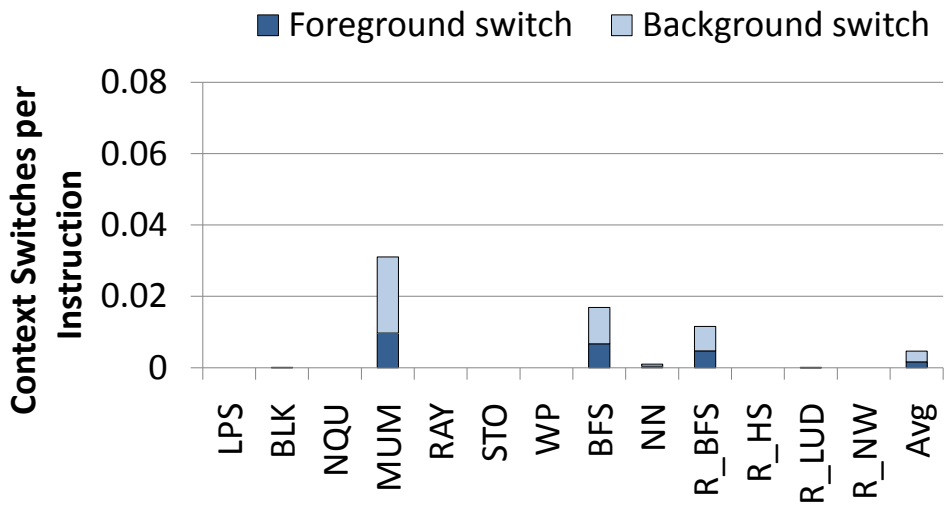
Retention time is another important consideration for a DRAM-based memory. Larger storage capacitors allow a longer retention time, but cost more energy on a context switch and take up more silicon area. For our hybrid SRAM-DRAM cell, we chose a 7.5fF capacitor with 2.3 ms retention time. In terms of GPU core cycles, this retention time presents over 1.4 million cycles.

4.4.3 GPU Performance

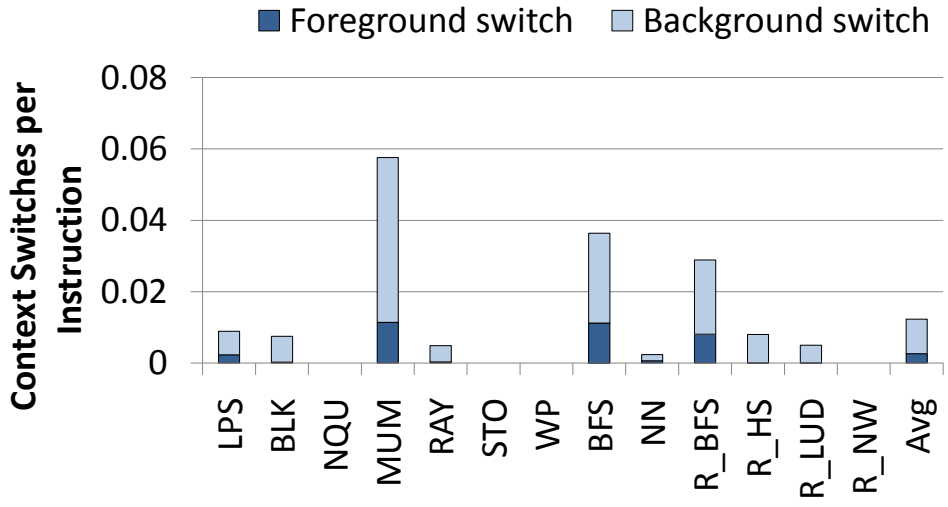
Figure 4.7 shows the performance of hybrid register files for a different number of contexts. The performance is normalized to the baseline SRAM register file. The figure shows that the performance overhead increases with more DRAM contexts per cell. This trend is expected as a fewer number of contexts implies more warps for each context, which leads to fewer context switches. Even with context switches, the overall performance loss is quite low for the 2 and 4-context configurations: 0.5% and 1.4%, respectively. On the other hand, the 8-context configuration yields high performance overheads in several benchmarks due to a greater number of context switches. The 8-context configuration shows an average of 13.3% performance loss. Note Gmean denotes geometric mean for all figures in this section.

Overall, we found that there are two main sources of performance overheads in hybrid register files: additional context switches and changes in warp scheduling. Most of performance overheads can be explained with the number of context switches while some benchmarks suffer from scheduling restrictions imposed by contexts. The following paragraphs discuss these two factors in more detail.

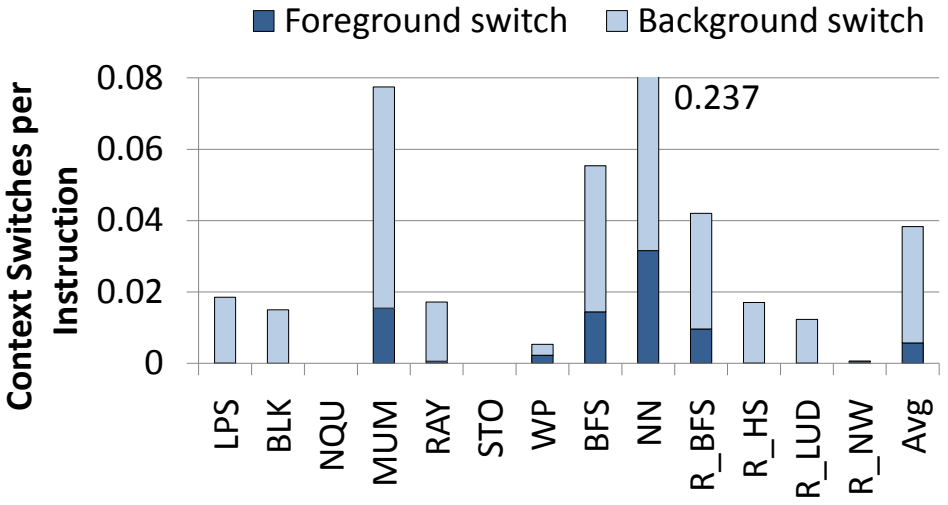
Figure 4.8 shows the frequency of context switches for the three context configurations. In the figure, we categorize context switches as either background or foreground. The background switches happen in the standby sub-bank while a pipeline continues using the other sub-bank. Therefore, a background switch latency can be hidden either partially or entirely depending on the number of ready warps. On the other hand, the foreground switches happen when there is no ready warp available in another sub-bank. It is important to note that



(a) 2-context register file



(b) 4-context register file



(c) 8-context register file

Figure 4.8: Context switch frequency (the number of context switches per instruction).

not all visible context switches cause additional pipeline stalls because context switches may overlap with other types of stalls. For example, if a pipeline is stalled at the fetch stage due to an instruction cache miss, even a foreground context switch can be effectively masked.

The figure illustrates that context switches happen more frequently for register files with more contexts, which is expected. However, context switch ratios are quite low in general. In fact, NQU and STO only use a small subset of registers and do not need any context switch across all three memory configurations. Also, while the overall context switch ratio increases, most context switches happen in the background. For example, most benchmarks have about the same number of foreground context switches for 2-context and 4-context configurations. As a result, most benchmarks except for MUM and RAY show negligible performance overheads for 2-context and 4-context configurations. MUM is the one with the highest context switch ratio. RAY is an outlier that we will discuss in detail later. It also appears that foreground context switches can be masked by other kinds of pipeline stalls as BFS and R_BFS do not show much slowdown even with noticeable context switch ratios.

The 8-context configuration in Figure 4.8(c) shows an interesting trend. There are several benchmarks, namely LPS, BLK, RAY, WP, and R_HS, which show significant performance degradation in the range of 10% to 30% even though their context switches are mostly categorized as background. This performance degradation comes from two sources. First, because there are only 2 warps per context in the 8-context configuration, even background switches can only partially hide the 3-cycle context switch latency. Also, these benchmarks show very high baseline IPCs in high hundreds (see Table 4.2). As a result, even

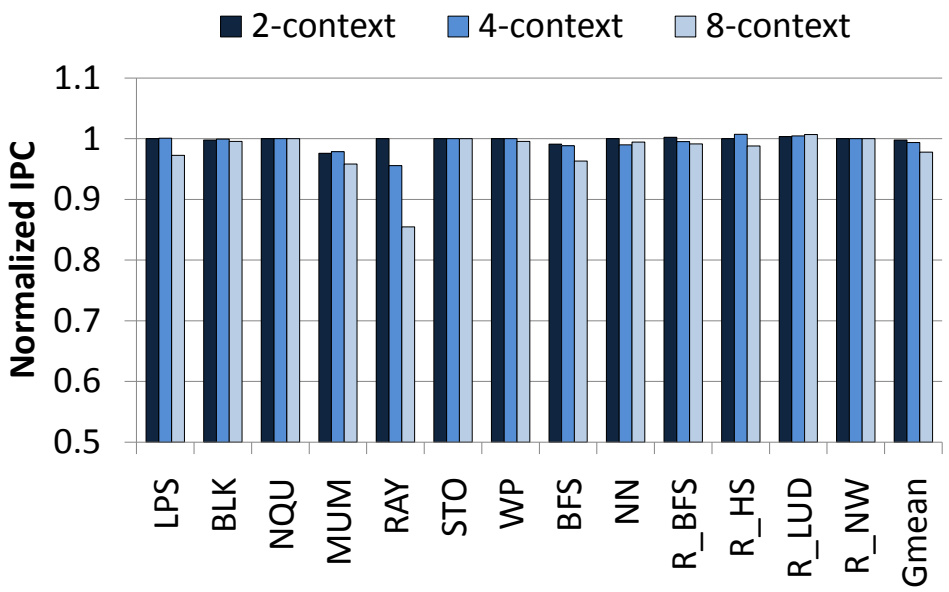


Figure 4.9: Performance comparisons for hybrid register files (2, 4, and 8 contexts) with 2048 threads per core, under 3 cycle context switch latency and 32 executions units per core. The performance is normalized to the baseline SRAM register file.

a small number of pipeline stalls significantly degrades the performance.

Figure 4.9 shows the performance overheads when the total number of threads per core is increased from 1024 to 2048. As shown in the figure, all benchmarks that previously had high performance overheads for the 8-context configuration show much lower overheads. In fact, the performance overhead is comparable to that of the 4-context case with 1024 threads per core. This shows that the number of warps per context is an important factor in performance overheads. With 1024 threads per core, the 4-context configuration has 4 warps per context, which is the same for the 8-context configuration with 2048 threads per core. Overall, with 2048 threads per core, the performance loss is 0.2%, 0.6%, and 2.2% for the 2, 4 and 8-context configurations respectively.

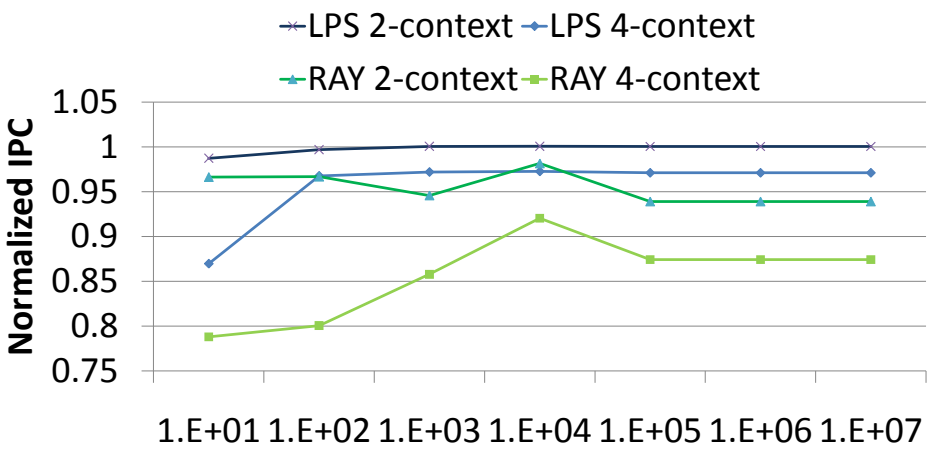


Figure 4.10: Scheduling trade-offs between fairness and context switches.

RAY shows a noticeable performance degradation although it incurs no context switches under the 2-context configuration and infrequent switches under the 4-context case. In this case, we found that the scheduling is the main source of the overhead. RAY only uses 12 warps per core, which results in an imbalance in the number of warps per context: 8 and 4 in the 2-context case. The greedy scheduling algorithm that tries to reduce context switches tends to favor the larger context. Figure 4.10 shows the performance of RAY and LPS as we vary the scheduler counter threshold that determines how long one context can run without being forced to context switch. For RAY, the figure clearly shows the tension between fairness and context switches. Small thresholds hurt performance by increasing the number of context switches. Large thresholds also result in non-optimal performance because only a subset of warps tend to finish early. This scheduling anomaly, however, is rare and the rest of the benchmarks behave much like LPS, which is relatively insensitive to the scheduler counter threshold.

We have also analyzed the performance with different context switching la-

tencies. In general, a higher context switch latency increases performance overheads, especially for benchmarks with noticeable context switch ratios. With a higher latency, background context switches are less likely to be hidden, and the foreground context switches would cause more stalls. For the 5 cycle switching latency, the performance loss is 0.5%, 7.8% and 22.6% for the 2, 4 and 8-context configurations respectively.

We have also studied the effect of a fewer number of execution units by reducing the number of execution units per core to 16. This effectively doubles the execution latency to 2 cycles per warp instead of 1. With a longer warp execution latency, there is a larger window to perform a background context switching, which in turn improves performance. The performance improvements are especially significant in the 8-context configuration, which cannot hide a context switch in the default configuration. With the 2-cycle execution latency, the average performance loss becomes 0.1%, 1.4% and 3.2% for the 2, 4 and 8-context configurations respectively.

DRAM refresh is another potential source of overheads as they may incur additional context switches. However, we found that the refresh operation almost never happens in practice because contexts are scheduled frequently enough. The 2.3 ms refresh time allows a window of 1,400,000 cycles before a refresh is needed. Simulation statistics indicate that contexts are scheduled again well before this window is met. In the worst case, a context remains dormant for 30,000 cycles, and the average case was below 1,000 cycles.

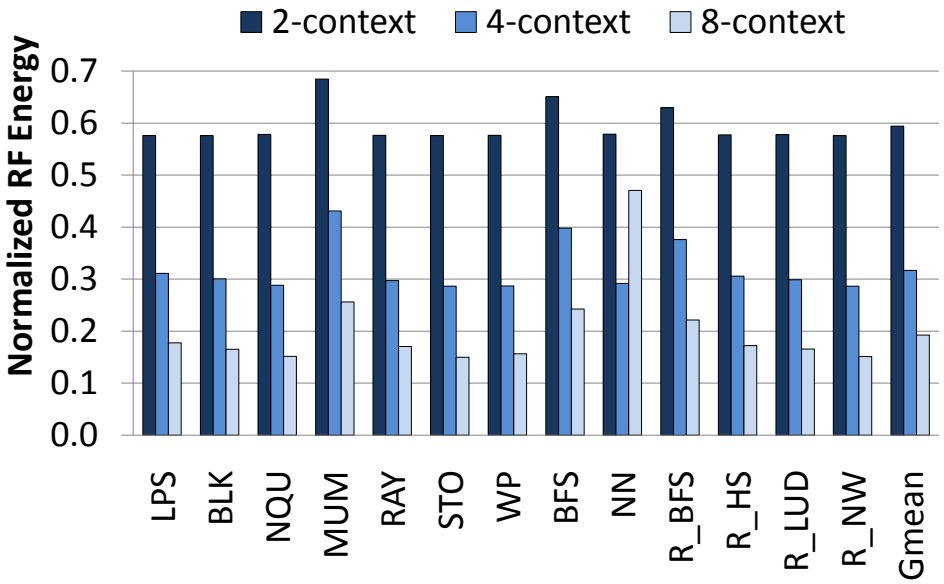


Figure 4.11: Normalized energy consumption for hybrid register files (2, 4, and 8 contexts). The results are normalized to the baseline SRAM register files.

4.4.4 Register File Energy Consumption

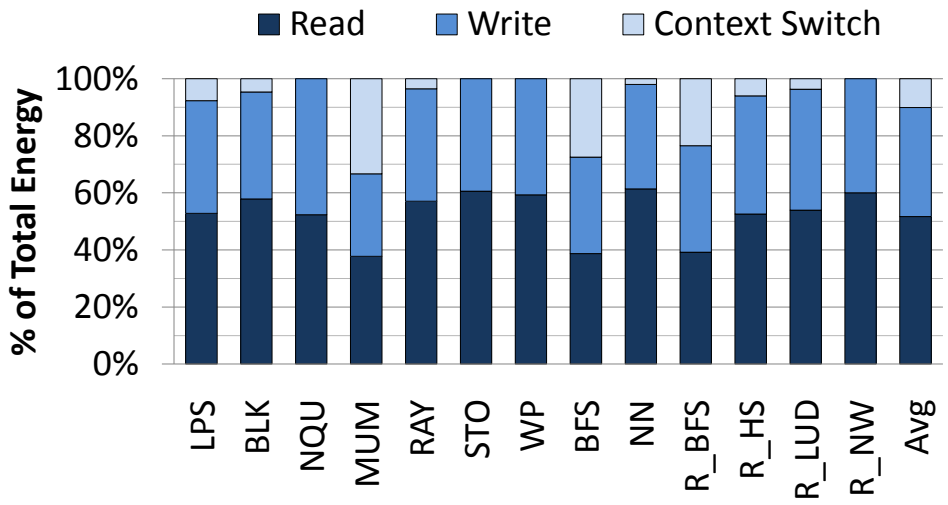
Figure 4.11 shows the estimates of register file energy consumption for the hybrid memory configurations. The results are normalized to the baseline SRAM array. Hybrid memory arrays show a significant reduction in total register file energy compared to the baseline SRAM array. As we increase the number of contexts for hybrid memory, the total energy continues to drop. The energy reduction can be attributed to the reduced read and write energy of the physically smaller hybrid SRAM-DRAM arrays as seen in Table 4.4. This result indicates that the energy savings on regular read and write operations are more significant than the additional energy consumption in context switches. Overall, the register file using 4-context hybrid memory arrays will consume roughly 68% less energy than the SRAM register file.

Figure 4.12 shows the energy breakdown of the hybrid memories for the 4 and 8-context cases. As we increase the number of contexts, more frequent context switching is required, and the percentage of the context switch energy in the total register file energy increases correspondingly. For NN, the figures show that an increase in context switches may outweigh the reduced energy in read and write operations. The energy consumption of NN in the 8-context configuration is higher than that of the 4-context configuration. We can also see that benchmarks with more frequent context switches, such as MUM and BFS, show less energy savings.

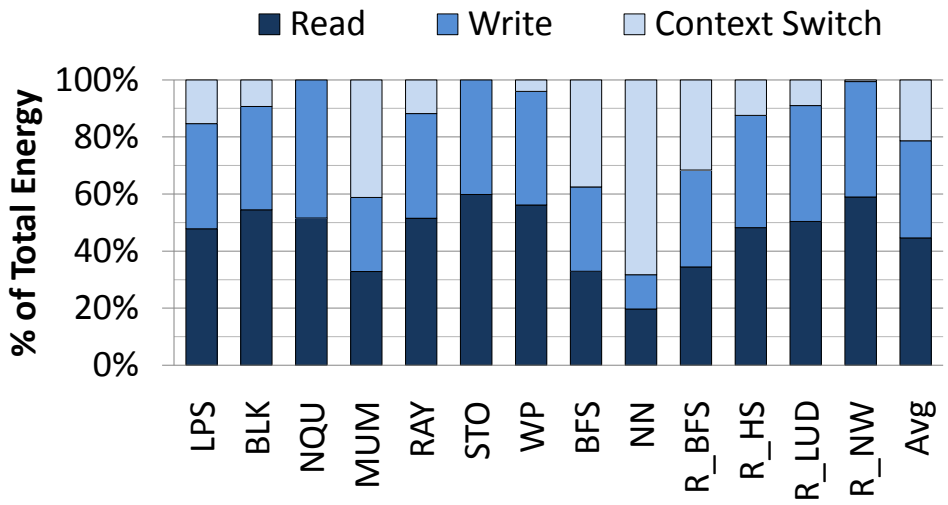
We also studied the leakage power in memory arrays as shown in Table 4.4. As the hybrid memories use less SRAM cells for the same storage, they consume only 74%, 37%, and 23% of the leakage of an SRAM array with the same capacity for 2-context, 4-context, and 8-context configurations. However, the leakage energy is not significant for the overall energy as it is orders of magnitude smaller than others.

4.5 Related Work

Embedded DRAM technologies. Embedded DRAM (eDRAM) technologies that are compatible with logic processes have been extensively studied in recent years, especially with the growing gap between on-chip and off-chip bandwidths and the emergence of multimedia applications. In fact, most of the major foundries offer eDRAM as a part of their standard embedded memory package. For example, our hybrid memory design is based on published numbers for IBM's eDRAM based on trench capacitors [29, 5]. Similarly, UMC offers



(a) 4-context register file



(b) 8-context register file

Figure 4.12: Register file energy breakdown for selected benchmarks.

eDRAM with trench capacitors in 90 and 65nm processes [49]. TSMC uses a Metal-Insulator-Metal (MIM) capacitor in their eDRAM offering, which is available from 0.18μ to 40nm processes, and reports two to three times better density compared to the 6-T SRAM [48]. Our hybrid memory cell design does not depend on a particular eDRAM technology and will be applicable to both trench and MIM capacitors.

Use of embedded DRAM. As eDRAM technologies mature, many recent commercial products utilize an eDRAM array as a high-density on-chip memory. For example, IBM Power4 and Power5 implement their L2 caches with the logic-based eDRAM technology [47, 40]. The Power7 microprocessor also includes an eDRAM L3 cache [54]. IBM also used eDRAM in the network data router [16]. As a mass-market product, Microsoft's Xbox360 uses eDRAM on the main processor die [41]. While the proposed hybrid memory also uses the eDRAM technology, this work focuses on integrating SRAM and DRAM together as a hybrid where as the previous examples simply use eDRAM as a discrete memory array.

SRAM-DRAM hybrid. Recently, Valero et al. proposed a SRAM-DRAM hybrid macrocell that is designed to implement first level data caches [50]. While the high-level approach to combine SRAM and DRAM technologies is similar to our hybrid memory, their cell design is quite different from ours and not efficient for a register file. More specifically, their macrocell stores n-bits using one 6T SRAM cell and $(n-1)$ 1T1C DRAM cells. Both SRAM and DRAM cells are accessible externally and data can only transfer internally from SRAM to DRAM. On the other hand, our hybrid memory can internally move data between SRAM and DRAM while only allowing access through SRAM cells.

In addition to the difference in the memory cell designs, this work also investigates architectural mechanisms to enable efficient use of hybrid memory in register files for fine-grained multi-threading.

Multi-context memory. Register files in multi-threaded architectures commonly store registers from multiple threads. In this sense, these register files are also *multi-context*. However, we use the term multi-context memory to indicate that memory is partitioned and only one partition (context) can be accessed at a time.

Conceptually, our definition of a multi-context memory is almost identical to the way register windows function in RISC I [36] or SPARC ISA. In fact, the register windows also allow efficient implementations through combining multiple memory technologies. For example, the register file in the UltraSPARC T1 microprocessor is composed of a multi-ported register file backed by a more compact SRAM array [22]. The current register window is accessible via the register file, whereas other register windows are stored in the SRAM until needed. While the high-level idea of combining different memory structures in register file is similar to register windows, our work studies hybrid cells that combine SRAM and DRAM rather than two types of SRAMs. Also, context switches in our hybrid memory are quite frequent and managed by hardware whereas typical register windows are managed in software and switched rather infrequently on function calls and returns. We believe that our hybrid memory design is applicable to register windows as well even though DRAM refresh will need to be handled more carefully.

4.6 Conclusion

This chapter presented SRAM-DRAM hybrid memory with a main application to an efficient implementation of multi-threaded register files. We laid out the proposed memory cell and arrays in an IBM 65nm process with an estimated area for an embedded DRAM cell. Thanks to the compact eDRAM cells, the hybrid memory arrays with multiple DRAM contexts provide significant area and energy savings compared to the SRAM array with the same capacity. This hybrid memory configuration is shown to be a nice match for a highly multi-threaded register files in GPUs that need to hold a large number of registers but only access a small part at a time. We modified a GPU pipeline and a scheduler for the hybrid register files. Simulations show that it is possible, with 4-context SRAM-DRAM hybrid arrays, to achieve 38% area and 68% energy savings on average with no appreciable loss (1.4%) in performance.

While we focused on reducing the area and energy of register files, we believe that the hybrid memory can be applied in many other ways and areas. For example, a hybrid memory can be used to increase register file size while remaining within fixed area and energy budgets. The hybrid memory can also provide near-instant checkpointing capabilities or improve the efficiency of other memory structures such as cache.

CHAPTER 5

HYBRID MEMORY FOR ENERGY-EFFICIENT CACHES USING COMPRESSION

5.1 Introduction

This chapter describes a hybrid memory designed to reduce cache energy usage using a combination of SRAM and another non-volatile memory (NVM), in this case, MRAM. The cache uses compression to leverage the strengths of each type of memory.

Instead of a monolithic SRAM or monolithic MRAM cache, a hybrid cache of the same capacity is constructed with a portion of the cache being SRAM and the remaining portion as MRAM. The hybrid cache reduces static energy versus an SRAM cache by using a smaller physical amount of SRAM and replacing a large portion of the cache with MRAM, which has low static energy. To address the expensive write energy of an MRAM-only cache, the hybrid cache uses compression: writes that would normally be written to the MRAM are converted to a smaller representation that is then only written to the SRAM portion of the cache.

5.2 Motivation

Cache energy reduction is an important research problem [30]. Because caches occupy a large percentage of die area, and are also critical to performance, caches are carefully optimized to maximize capacity while remaining energy-

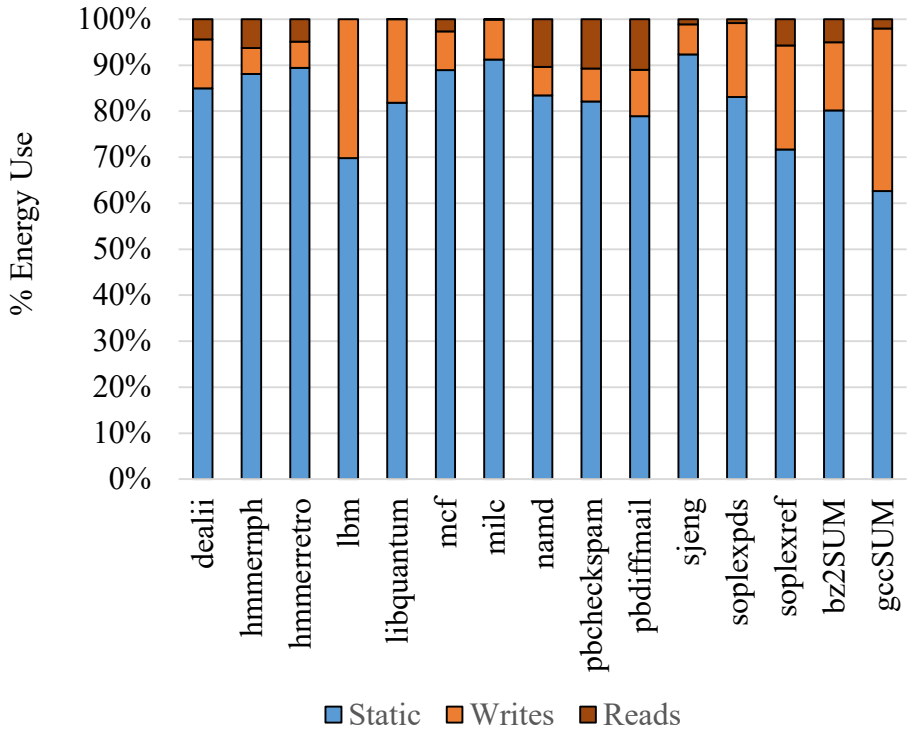


Figure 5.1: SRAM L2 cache energy usage by function, for benchmarks (described in Table 5.1).

efficient. Design tradeoffs are made based on power and energy concerns, as well as area.

Traditional SRAM caches have their energy usage dominated by static energy. An example of this, using data for a 256kB L2 from our evaluation (Section 5.5), is shown in Figure 5.1. The vast majority of the energy used for our benchmarks and performance model is spent in static leakage. To reduce static energy usage, architects have tried many ideas, such as drowsy caches and other power-reducing techniques. These solutions are effective, but the inherent high static energy consumption of SRAM is a persistent thorn to tackle.

To move away from the limitations of SRAM itself, recent proposals have

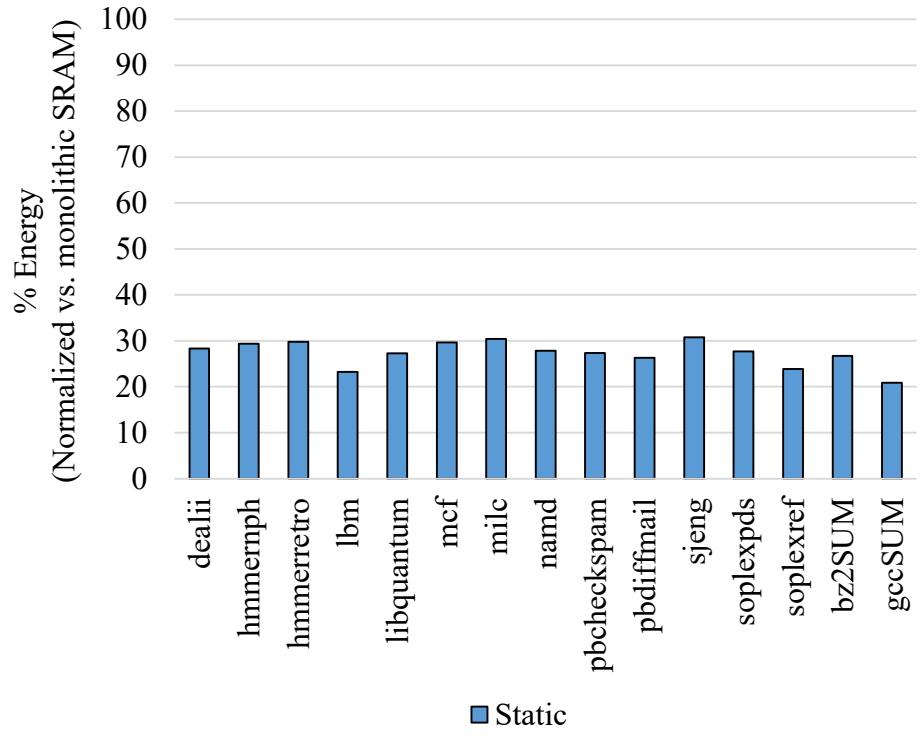


Figure 5.2: MRAM static cache energy usage, normalized to sram energy, for benchmarks (described in Table 5.1).

used newer non-volatile memory technologies in place of SRAM. These technologies have appealing characteristics such as low static energy, and non-volatility allows them to retain their data beyond a power cycle. They can also have a smaller footprint than SRAM, which allows denser construction. Replacing an SRAM with MRAM of equivalent capacity reduces static energy usage by a large amount, as shown in Figure 5.2. The figure shows the static energy component of an MRAM of equivalent capacity (256kB) as the SRAM shown in Figure 5.1. As the bars are normalized to the SRAM energy usage, we see that the static energy of an MRAM cache is much less than an SRAM cache's.

However, monolithic MRAM caches have disadvantages of their own. Non-

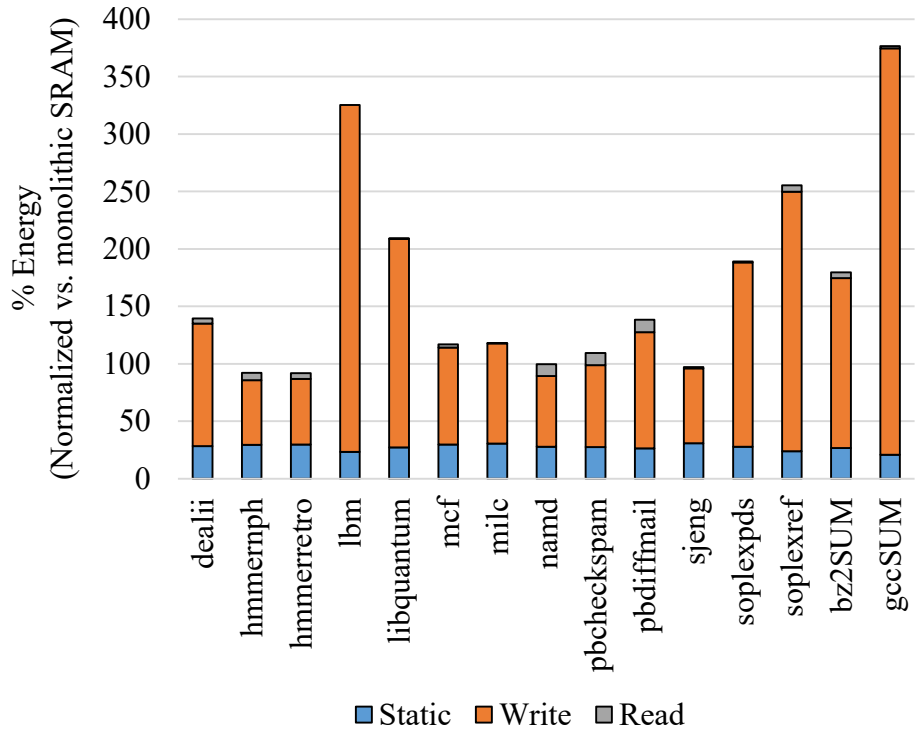


Figure 5.3: MRAM total cache energy usage, normalized to sram energy, for benchmarks (described in Table 5.1).

volatile memories have high write energy and latency, and also have to deal with limited write endurance. The high write energy can negate or even surpass the energy saved from the lower static energy. Figure 5.3 shows total MRAM cache energy usage, split into static, write, and read components. Note that the y-axis of this graph is enlarged – it goes to 400% of a monolithic SRAM's energy. The massive write energies of MRAM, when unmitigated, cause the overall cache energy expenditure to balloon well past that of static-energy hungry SRAM.

Consequently, a lot of work has focused on ways to mitigate these write disadvantages. A cache made entirely of a non-volatile technology, while ex-

periencing much lessened static energy usage, must contend with these write limitations.

Architects have proposed methods to address these write limitations by reducing the amount of bits that are written to the cache itself [56, 17, 1], often using compression schemes. These schemes are effective at reducing write energy of the non-volatile memory caches.

A hybrid cache is potentially able to save more energy than either monolithic solution. Note that the characteristics of SRAM and non-volatile memory complement each other. While SRAM has high static energy, non-volatile memory has low static energy. Non-volatile memory's weakness, high write energy and latency, is not shared by SRAM. If a cache could be designed that writes preferentially to SRAM (avoiding non-volatile memory writes), while minimizing SRAM area (minimizing static energy), then a hybrid solution, by converting non-volatile writes into cheaper SRAM writes, could be a viable design point. This would be a unique characteristic of a hybrid cache as well.

To perform this conversion, a compression scheme could be used. If an incoming value to be written to cache is compressed, it could be completely stored in the SRAM portion of the cache, bypassing the non-volatile section.

Note that the energy savings of the hybrid cache relies on the energy saved by converting non-volatile writes to SRAM writes is greater than the energy used by the SRAM portion to store it (mainly, static energy).

In the next section, we discuss the particulars of the hybrid cache designed to exploit the above idea. We go over the structure in Section 5.3, non-volatile memory choice in Section 5.3, and the compression algorithm in Section 5.3.

5.3 Proposed Hybrid Memory Cache

The proposed hybrid cache architecture would reduce energy usage for a cache of the same capacity by:

- Using SRAM and NVM arrays in parallel. An SRAM array with less capacity *reduces static energy consumption,* while the remainder of the cache capacity will be serviced by a non-volatile memory, with low static energy.
- To mitigate the high write energies of the non-volatile memory, compression is used. Commonly written cache lines would be compressed and stored in the smaller SRAM; completely *obviating writes to the non-volatile section.*

Operationally, a cache line is split into a section serviced by the SRAM, and a section serviced by a non-volatile memory, as shown in Figure 5.4. Because this only affects the data array, the tag array, address decoders, and sense amps remain untouched. The tag array is augmented with additional bits to indicate whether values in the cache line are compressed or not.

When a cache line enters the cache, it is checked to see if it can be compressed. If so, the line is stored only in the SRAM, and the non-volatile memory remains untouched. If a cache line is not compressed, then the entire cache line (SRAM + NVM) is used. Figure 5.4 shows this graphically.

The SRAM section is sized to be just large enough to fit the largest possible codeword used for compression. This ensures that, when the tags for the cache line are read, if the value is compressed, only the SRAM must be read. It is possible to size the SRAM section differently, though. For example, if the SRAM

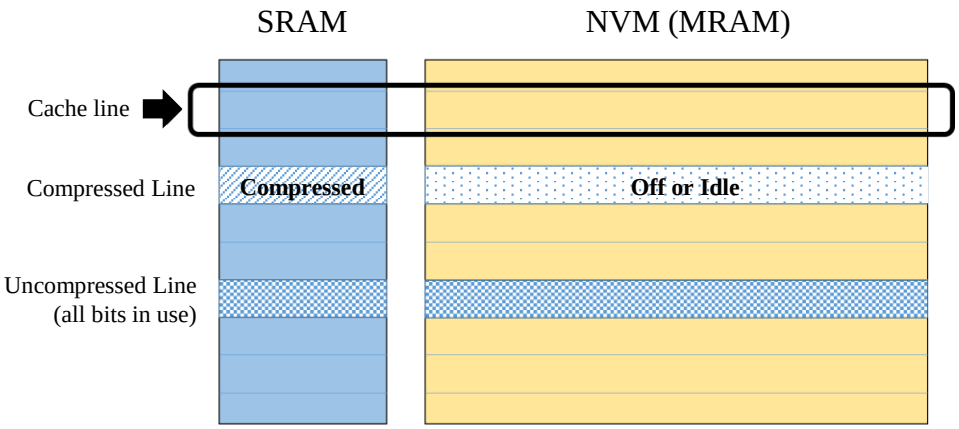


Figure 5.4: Detail of compressed and uncompressed cache line placement in hybrid cache.

section is sized smaller than the maximum codeword, a compressed value could be partially written to both SRAM and NVM. A shorter codeword that fits into the SRAM section would still avoid writing to the NVM. The extra write energy used in longer codewords could be offset by the lower static energy of the SRAM section.

Compression logic would need to inspect every incoming cache line to be written. Decompression logic would be needed just for the output coming from the SRAM array, as no compressed values are stored in the NVM section.

Taken together, the cache block diagram is shown in Figure 5.5.

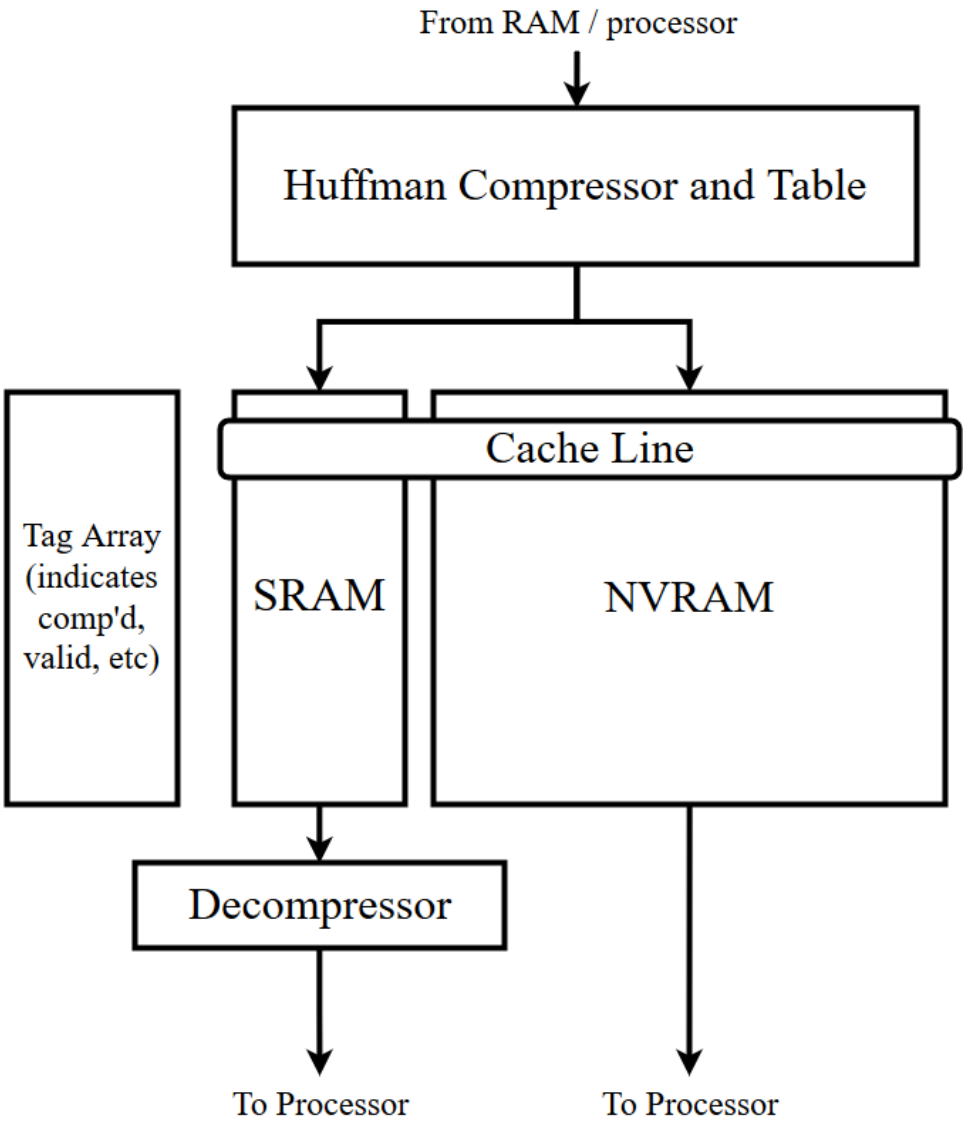


Figure 5.5: Proposed hybrid cache block diagram, using Huffman compression.

Non-Volatile Memory Choice

For cache usage, there are only a couple of choices for a suitable non-volatile memory. Because of the high frequency of writes, only a couple non-volatile memory technologies have the required endurance: Magneto-resistive memory (MRAM), and resistive memory (RRAM). While their estimated endurances are high enough to act as a cache technology, they are still not as high as SRAM and DRAM endurances.

We study MRAM because it has received the most attention from other architects, and several comparison studies versus SRAM have been published [53, 7, 55, 42, 17, 9]. This gives us valuable reference points from which to compare.

RRAM has received less attention, however, at the time of writing, it is becoming more attractive as a leading non-volatile memory candidate, due to improved fabrication techniques and more favorable projected scaling characteristics.

Compression Algorithm

Cache compression is a well studied topic. For our hybrid cache, we want to select an algorithm that has minimal performance penalty, and with a high compression ratio, in order to reduce the number of writes to the non-volatile memory.

We considered several algorithms:

- Zero Compression. This scheme replaces zero values with a 1-bit tag indicating the value is zero.
- Frequent Pattern Compression. This scheme codes frequent values into varying size codewords, and is a common choice in cache compression [1].
- Data-Comparison-Write (Diff). This is not a compression scheme per-se. On an incoming write, the existing value is compared with the new value. Only bits that change are written to the memory [56].
- Dynamic Huffman Encoding. A recent state-of-the-art scheme which samples incoming values and dynamically generates a Huffman code for the most frequent ones [2].

In our evaluation section we describe the reductions each scheme is able to achieve. At final reckoning, we choose the dynamic Huffman scheme proposed by Arelakis [2]. We modify their scheme to only count written values, instead of both read and written values in the original paper (their original goal is not to minimize writes, but compress the most frequently used values in order to fit more data within the cache).

For ease of reference, Arelakis's scheme, with our modifications, is briefly described here. The scheme involves the addition of 3 tables and a barrel shifter, as well as some comparison logic.

During the sampling period, values incoming are tallied in one of the tables (value counting table). After the period is over, a software routine is called, which builds the Huffman code, and then places translations from uncompressed to compressed values (and vice versa) in an encoding table (and de-

coding table). The value counting table is not used after sampling.

The Huffman code can be built at varying granularities; 4 bytes (16 bit) is deemed most effective by their work, and this was confirmed in our evaluations. This means that there are up to 16 Huffman codes per 64 byte line.

An incoming value to be written is sent to the barrel shifter and comparison logic, which finds a corresponding match in the encoding table if one exists. If it finds one, then the table is accessed and the compressed Huffman codeword is sent to the SRAM to be written, and the compression tag set. Otherwise, the value to be written is split and sent to both the SRAM and MRAM.

If an incoming read is found to be compressed, only the SRAM is read and the compressed value then sent to the decoding table. The decoding table returns the uncompressed value, which is then sent beyond the cache to its consumer. If the read value is uncompressed, then the SRAM and MRAM contents of the line are simply read and sent to the consumer. The SRAM portion would bypass the decompressor.

5.4 Evaluation Methodology

In this section we describe how the hybrid cache is evaluated.

5.4.1 Benchmarks, CPU Parameters, and Trace Collection

We collect traces using the Intel Pin tool[27]. We run various SPEC2006 benchmarks, fast-forwarded 2 billion instructions to reach non-initialization code, and

CPU Parameters and Benchmarks Tested		
CPU Speed	3 GHz	
Caches	Configuration	Access Latency
L1 data / inst	32kB, 8-way	1 cycle
L2	256kB, 8-way	10 cycle
L3	2MB, 16-way	25 cycle
Benchmarks	bz2*, gcc*, dealii, hmmernph, hmmerretro, lbm libquantum, mcf, milc, namd, pbcheckspam pbdiffmail, sjeng, soplexpds, soplexref	

Table 5.1: Simulation CPU parameters and benchmarks.

then run for 250 million instructions. The benchmarks are listed in Table 5.1. For the different benchmark inputs for bz2 and gcc, we sum them into one result bar for conciseness.

The traces contain cache memory access patterns as well as the data written into the cache. It is parsed and analyzed to evaluate the compression schemes, as well as provide event counts for energy modeling. Reads and writes are tracked and used as events in our model. A read is any operation requiring data to be read from the cache directly. A write is any operation that requires data to be written to the cache. Note that this does not directly correspond to loads and stores at the instruction level. For example, a load that misses in a cache would be tallied as a write operation, as the missing data must be fetched and written to the cache. The read operation associated with a miss is not recorded, as the missing data is forwarded to the processor directly, instead of coming from the cache itself.

CPU parameters simulated are shown in Table 5.1.

Our cache of interest is the L2 cache, because it carries enough traffic over the number of instructions simulated for most benchmarks. L1 and L3 are not

treated as hybrid memories. L1 caches are performance sensitive; a compression scheme's additional latency affects cache access adversely. In addition, because L1 caches are smaller, there are less potential energy savings from a hybrid scheme. L3 caches are a good candidate for this scheme, however, for our benchmarks and runtime, there was not enough traffic generated to analyze meaningfully.

5.4.2 Compression Effectiveness

We would like to evaluate the effectiveness of the different compression schemes on reducing the number of writes. We test the zero compression, frequent pattern compression, and dynamic Huffman algorithms, described in Section 5.3.

The traces contain the actual data written to the cache. Each compression scheme is implemented and uses the trace data to tally the number of writes that occur.

5.4.3 Energy

For each benchmark, the energy of each cache configuration is modeled by combining the events tallied from traces with the energy per event. Events such as writes, reads, and hits and misses are counted. A detailed explanation of how events are counted can be found in Appendix A

Energy per event is modeled using estimates from CACTI, a tool used for

SRAM Energy Parameters			
Capacity	Read (J)	Write (J)	Static (Watts)
256 kB	1.60e-10	1.60e-10	3.00e-2
88 kB	3.30e-11	3.30e-11	7.70e-3

Table 5.2: Energy estimates from CACTI for SRAMs of 256kB and 88kB (1/3 size).

cache energy estimation [33]. CACTI models SRAM access times and energies for various configurations of caches and technologies. We use the smallest technology available, 32nm, and set CACTI to minimize area. For cache data arrays we use low static energy transistor models, and for tag access we use high performance models.

Energy overheads of the data-comparison-write (DCW) and Huffman schemes are modeled. DCW overhead consists of doing an additional read operation for every write. Huffman overheads include the energy use of the lookup tables and additional logic (modeled using CACTI).

SRAM Energy Modeling

The hybrid memory cache exploits the energy advantage of a smaller SRAM array. The smaller SRAM array has a reduced energy consumption because itit requires less area, reducing leakage and reducing bitline lengths. For the smaller SRAMs used in the hybrid memory, we set CACTI to use the same amount of tag bits as the full-size monolithic array, but to use a smaller set of data bits.

Table 5.2 shows the SRAM energy parameters we derived from CACTI estimates:

MRAM/SRAM Energy Parameter Ratios			
	Read	Write	Static
Minimum	0.5x	9x	0.11x
Maximum	2x	11x	0.33x
Used in Model	1x	10x	0.3x, 0.11x

Table 5.3: Typical ratios of MRAM/SRAM energies seen in literature.

MRAM Energy Modeling

To model MRAM, we used estimates derived from works that directly compared the energy usages of SRAM and MRAM arrays of equivalent capacities. There are several of these works, and the range of energies they report varies somewhat [53, 7, 55, 42, 17, 9].

The relevant parameters are the ratios in energy cost of the read, write, and static energies of equivalently sized SRAM and MRAM arrays. We tallied the range of ratios and constructed our energy model with reasonable estimates, testing the extreme ends of the collected parameters to ensure sanity as well.

Table 5.3 below gives the ratios we encountered in the literature and the ratio(s) we used in our energy model.

For the MRAM capacity to estimate, we use CACTI to estimate energy parameters for an SRAM of that capacity, and apply the ratios from the Table 5.3 to arrive at the final model parameters. We sized the MRAM at two-thirds the size of the 256kB L2 (176kB). In CACTI, we used an SRAM of size 176kB, and then calculated the corresponding MRAM estimates. The resulting numbers are shown below in Table 5.4.

MRAM Energy Parameters			
Capacity	Read (J)	Write (J)	Static (Watts)
256 kB	1.60e-10	1.60e-09	1.00e-02
176 kB	3.60e-11	3.60e-10	6.00e-03

Table 5.4: Energy estimates from CACTI and literature ratios of MRAM/SRAM energies seen in literature, for MRAMs of size 256kB and 176kB (2/3 size).

Static Energy Modeling

While we have static energy (expressed per time) estimates from our energy models above, to find total static energy, we need to estimate how long each benchmark has been running. To model the execution time of each benchmark, we use a straightforward in-order model, which stalls the processor on a cache or memory miss. We assume a completion rate of 1 cycle per non-memory instruction. This model will give estimates longer than an out-of-order machine.

We account for the additional latency of the compression scheme by adding 5 cycles to the latency of the L2 cache (15 instead of 10). This time corresponds to the estimated latency of the dynamic Huffman scheme's decompressor [2].

Energy Model Details

In Appendix A we give the details on how our combined energy model is constructed.

5.4.4 Endurance

Because non-volatile memories have limited endurances, reducing the number of writes to the memory will improve their lifetime. We track the number of writes to each bit and report the improvement in the number of writes to the most worn out bit.

5.4.5 Cache Configurations

We compare the following caches, with and without Huffman compression. They are sized to hold the same capacity (256kB in our experiments).

- a monolithic SRAM cache
- a monolithic MRAM cache
- a monolithic MRAM cache with Data-Comparison-Write (or Diff)
- a hybrid SRAM (1/3) + MRAM cache (2/3)

5.5 Evaluation Results

In this section we discuss the results of the compression and energy evaluations performed, for each of the cache configurations tested.

5.5.1 Compression Effectiveness

Figure 5.6 shows the number of bits written per benchmark depending on the compression scheme. Lower means the compression scheme was more effective at reducing writes. The data is normalized to bits written when no compression is used at all.

Looking at the results, it is clear that Huffman scheme does the best overall, though for some benchmarks it is about equal with zero compression. Some benchmarks are more amenable to compression than others.

The real result to note here, however, is that the difference scheme (Data-Comparison-Write) does the best by far. This scheme is very effective at reducing bit writes. It saves on average 90% across all benchmarks of the bits written compared to using no compression scheme. Huffman saves an average of 64%. Zero Compression and FPC do worse, with 51% and 48% saved on average.

The difference scheme can be applied in addition to using compression. In Figure 5.7 we see the results of applying the difference scheme to the FPC and Huffman compression schemes. In this graph, the number of bits written is normalized to the DCW scheme itself. Huffman + DCW performs quite well, saving an additional 33% more than DCW alone.

When testing sensitivity of the scheme to the Huffman table size, we found that a small table of 128 values is able to provide enough compression. Higher values did not appreciably reduce the number of bits written, and the increased size of the lookup tables cost more energy.

As the best compression scheme is Huffman + DCW, we choose this scheme

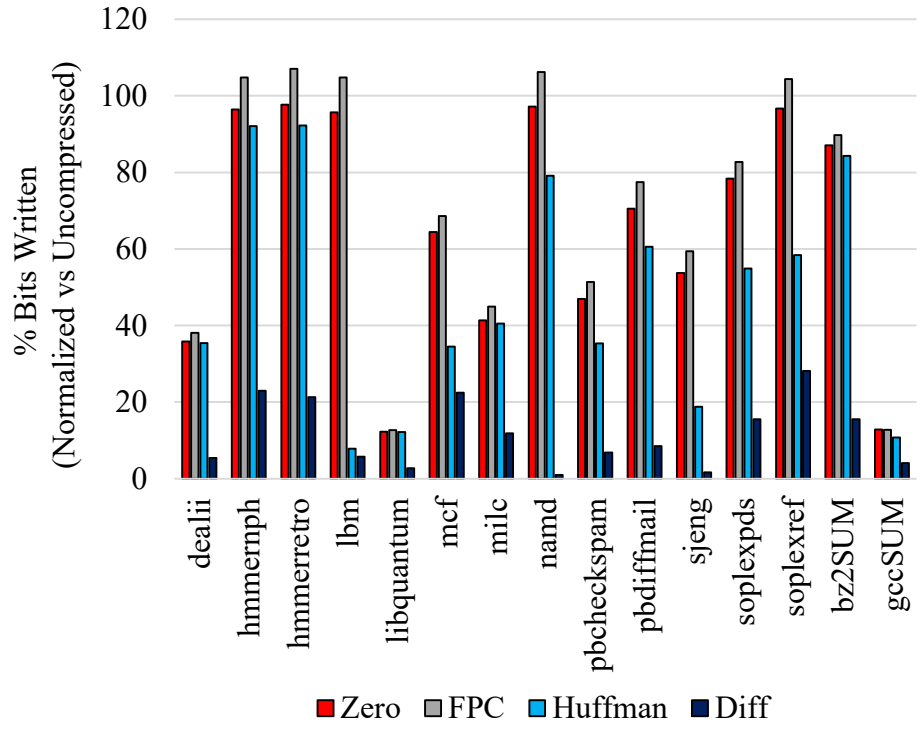


Figure 5.6: Compression effectiveness of zero, FPC, Huffman, and DCW schemes.

for the hybrid cache implementation, and will compare a hybrid cache with Huffman compression and difference comparison to monolithic caches with and without these schema.

5.5.2 Energy

This section compares the energy consumption for the hybrid cache versus the other cache configurations mentioned.

First, in Figure 5.8, we compare cache configurations without compression. The y-axis is normalized to the energy of a monolithic SRAM, and each bench-

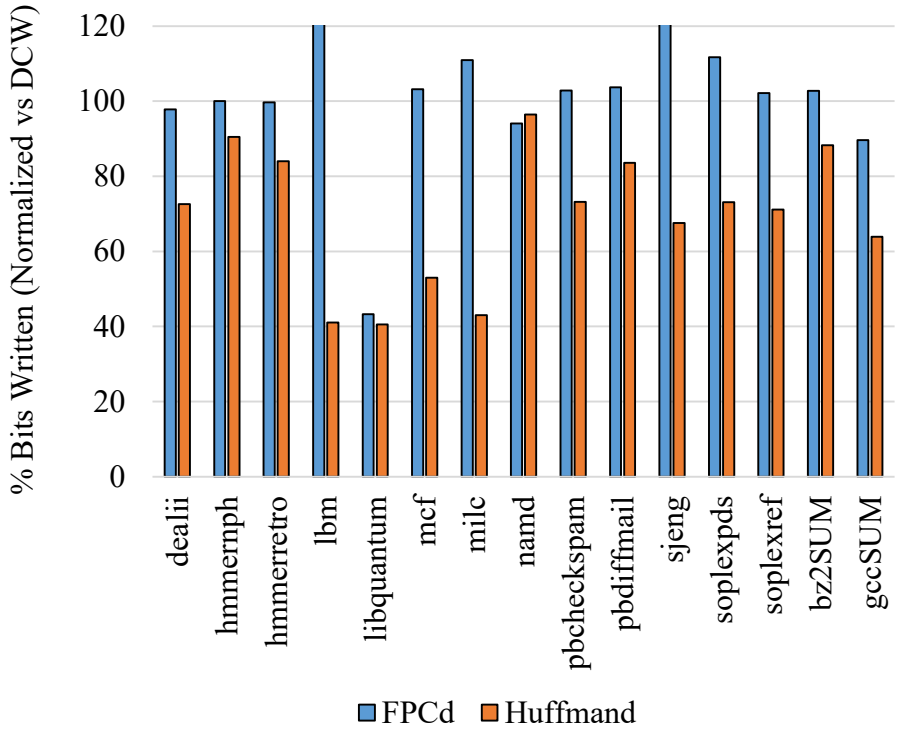


Figure 5.7: Compression effectiveness of Huffman + DCW (Huffmand) vs. FPC + DCW (FPCd), normalized to DCW alone.

mark has two bars: left is MRAM energy and right is hybrid configuration energy. Much less energy is spent on writes, because of the lower write energy of the lesser quantity of MRAM.

Second, in Figure 5.9 we compare 4 schemes. The y-axis is normalized to the monolithic SRAM energy for that benchmark. Each benchmark has 3 bars. The leftmost bar is a monolithic MRAM using only the DCW (difference) scheme. The center bar shows a monolithic MRAM using the Huffman scheme and DCW. Lastly, shown as the rightmost bar, we have the hybrid cache, using Huffman and DCW (HHD).

We can see that for some benchmarks all three schemes perform similarly,

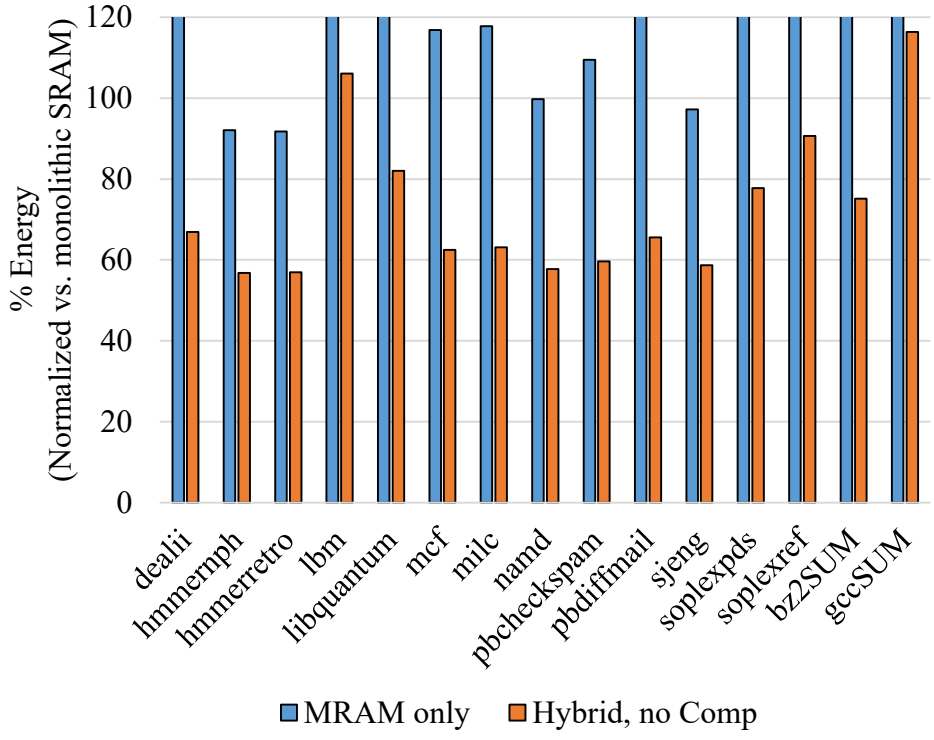


Figure 5.8: Energy usage of monolithic MRAM and hybrid caches without compression.

while for others, we see that the hybrid cache does much better than the monolithic MRAM caches (i.e. in soplex, bz2, gcc). The hybrid scheme works well for these benchmarks because it succeeds at converting MRAM writes into SRAM writes (i.e. high compressibility, especially vs. DCW). There are a few benchmarks with high compressibility that do not show as much energy advantage for the hybrid scheme. This is because they either do not have many writes in general, or that after the reduction in writes from the DCW and compression schemes, there are not enough writes to let the conversion have a noticable effect. It is good to note that even in these cases, the hybrid cache does not consume much more energy if any, than the monolithic MRAM and difference scheme. Figure 5.10 shows this in more detail, renormalizing the MRAM and

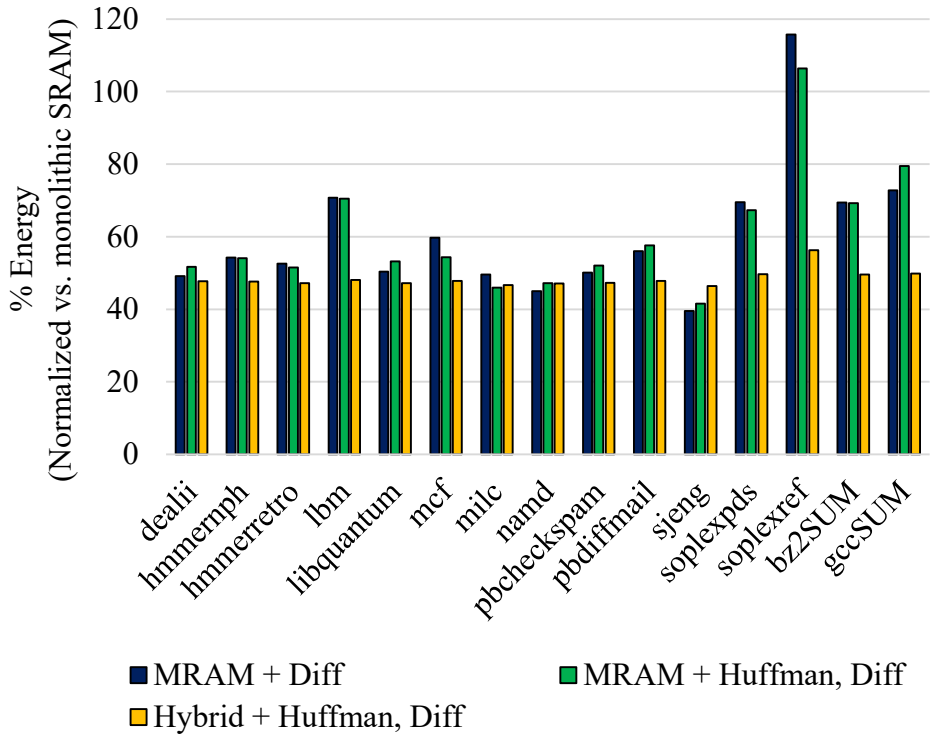


Figure 5.9: Energy usage of monolithic MRAM and hybrid caches with compression.

hybrid huffman schemes to the MRAM and difference scheme.

Compared to the monolithic SRAM, the hybrid memory consumes much less energy, saving an average of 52% across the benchmarks, with the range being 34% to 54%. Comparing to the monolithic MRAM and difference schema in Figure 5.10, the hybrid scheme saves -17% to 51% energy, with an average of 14% savings.

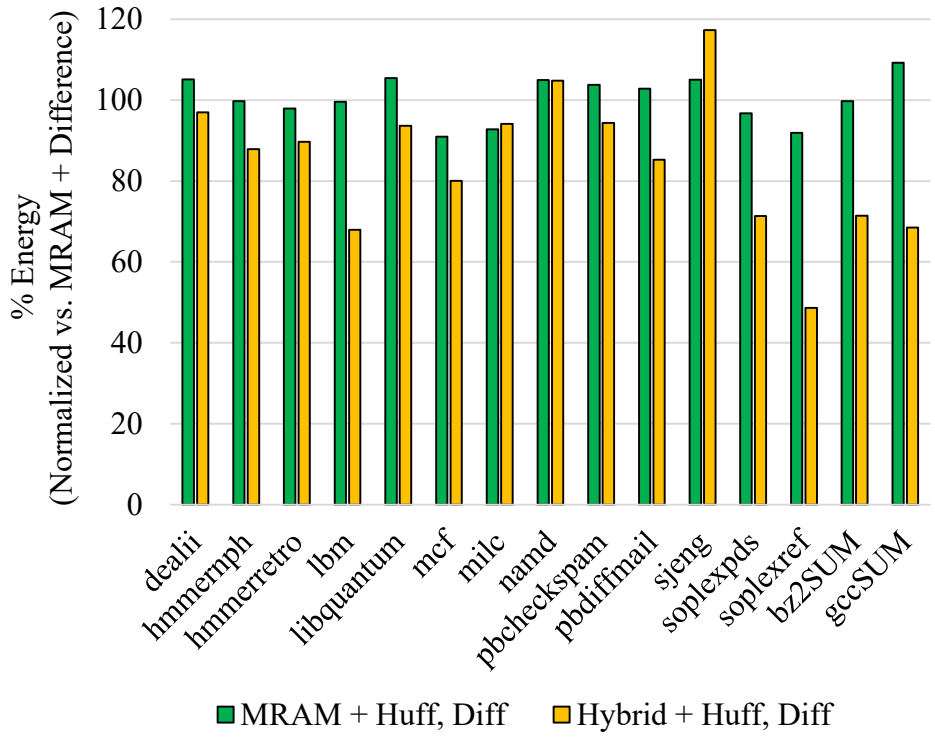


Figure 5.10: Energy usage of monolithic MRAM and hybrid caches with compression, normalized to MRAM + DCW.

Parameter Variation

The hybrid cache achieves its energy improvements because it saves static energy over a monolithic SRAM and write energy over a monolithic MRAM.

To explore the sensitivity of the hybrid cache, we ran the energy model with some other memory and architecture parameters, i.e. with faster performance, or with more conservative cache energy scaling (versus size). We explore these scenarios here. Overall, the hybrid cache is more consistent in terms of its energy savings than MRAM + DCW configurations, while still remaining better than or only slightly worse than them as well.

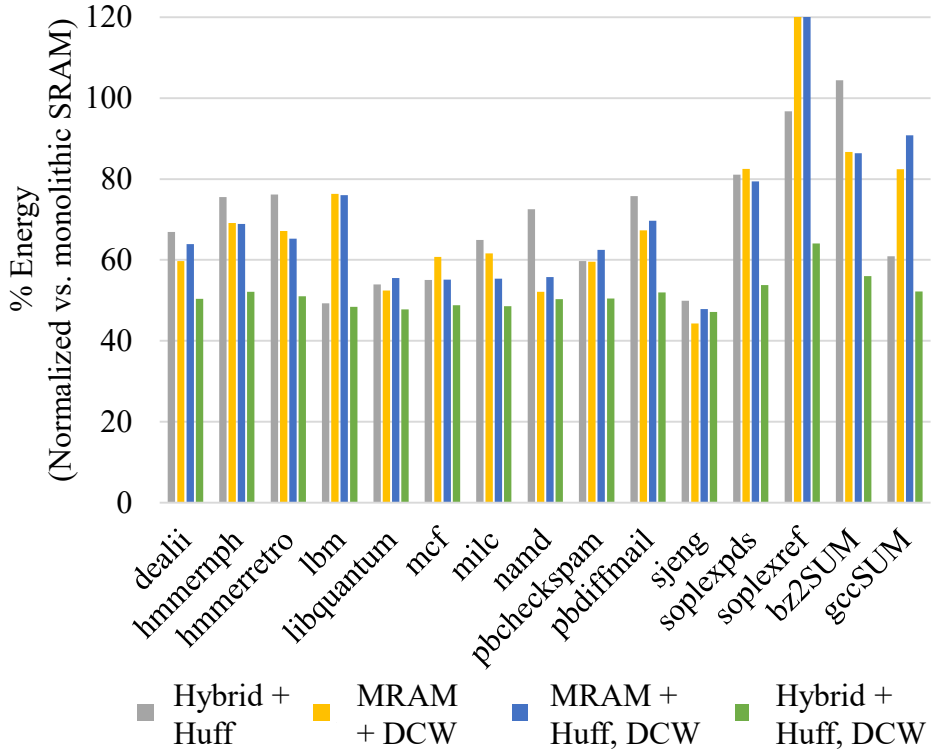


Figure 5.11: Energy usage of cache configurations, with a 50% faster processor.

In each of the following graphs, the energy is normalized to monolithic SRAM energy. The configurations are labeled as follows: 'Hybrid + Huff' is the hybrid cache with Huffman compression, 'MRAM + DCW' is the monolithic MRAM with DCW, 'MRAM + Huff, DCW' is the monolithic MRAM with Huffman and DCW. 'Hybrid + Huff, DCW' is the hybrid cache with Huffman compression and difference detection.

Figure 5.11 shows the energy results when using a 50% faster processor. This directly and only impacts the proportion of energy spent on static leakage, and benefits the monolithic SRAM configuration relatively more versus the other configurations.

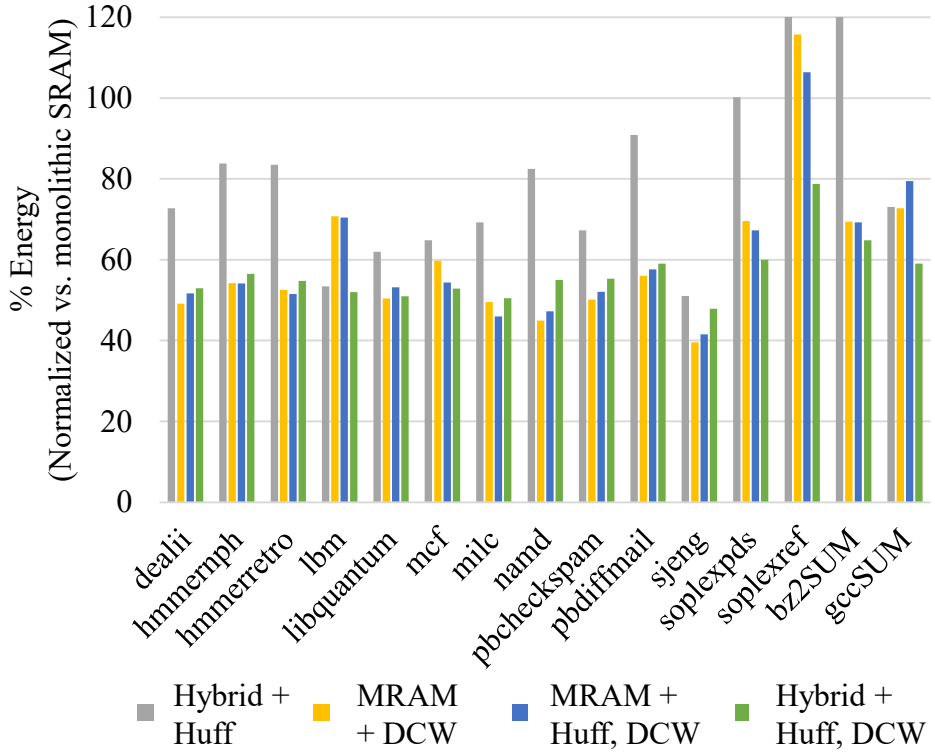


Figure 5.12: Energy usage of cache configurations, with linear cache energy scaling.

Overall, the hybrid cache with compression and difference does well even for faster processors. On average, it improves its gain against the MRAM solutions, being 22% better than MRAM + DCW, and because the faster performance aids the monolithic SRAM most, a slight reduction in gain is seen – a 49% avg improvement vs. monolithic SRAM. It is more consistent than the MRAM solutions as well, with a range of 18% to 54% savings. The MRAM + DCW cache ranges from -44% to 66% savings vs. a monolithic SRAM.

Figure 5.12 shows the scenario where we linearly scale the cache energy parameters with size, instead of using CACTI's estimates. This assumes that cache energy scaling is less than that estimated by CACTI's models, and should

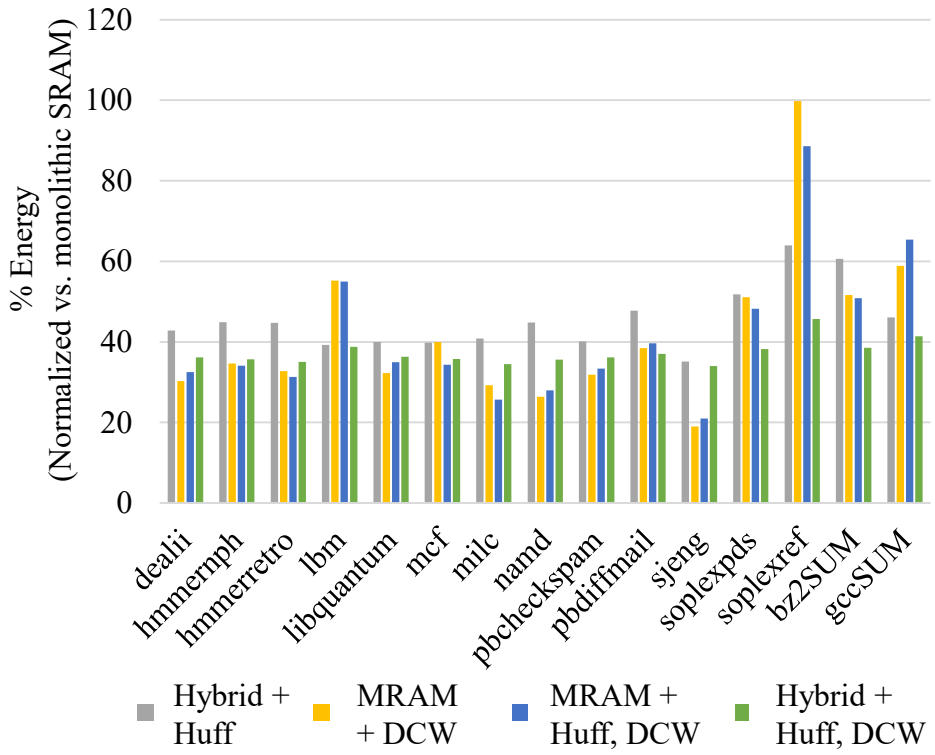


Figure 5.13: Energy usage of cache configurations, with MRAM using only 1/9 SRAM static energy.

place the hybrid scheme at a disadvantage, because it relies on exploiting the advantages of smaller physical arrays. Overall, the hybrid cache with Huffman and DCW still saves energy over monolithic MRAM caches with compression. The hybrid cache with compression exhibits less variation than the other schemes, and shows marked improvement still, over monolithic SRAM (on average, 47%). It does 1% better than the MRAM+DCW scheme, but exhibits less variation (-2% to 54% vs. -20% to 66%). It performs best in the write-heavy benchmarks that effectively use compression (i.e. soplexref), as expected.

Because non-volatile memories' most attractive advantage is lower static energy, we also tested the lower end of the static energy estimates, which had

MRAM using only one-ninth the static energy of an equivalent capacity SRAM. This would proportionally benefit the monolithic MRAM caches the most.

Figure 5.13 shows the results of this scenario. The hybrid cache with compression and DCW saves energy over monolithic SRAM, but loses on average to MRAM + DCW (the hybrid cache is, on average 6% worse in energy consumption vs. the MRAM + DCW cache). It saves energy over a monolithic SRAM cache, as would be expected, showing an average of 65% savings. As seen in the other scenarios, the hybrid cache is more consistent, with less variation, saving 47% to 67% across the benchmarks. MRAM + DCW savings over the monolithic SRAM cache range widely, from 0% to 88%.

When varying cache parameters, the hybrid cache with Huffman compression and DCW shows resiliency and can still deliver energy savings.

5.5.3 Endurance

Finally, we turn our attention to the question of endurance. For the following graphs, the y-axis shows the improvement in endurance. This is calculated as the number of writes without compression divided by the number of writes with compression. For example, if a bit in the array sees 100 writes without compression, and 25 writes when compression is on, then the improvement would be 100/25 = 4. Some egregious outliers have been removed (too few writes).

First, we compare the hybrid scheme with the Huffman compression and DCW scheme versus a cache without any compression or DCW. The results are shown in Figure 5.14. Predictably, the endurance is vastly improved, with

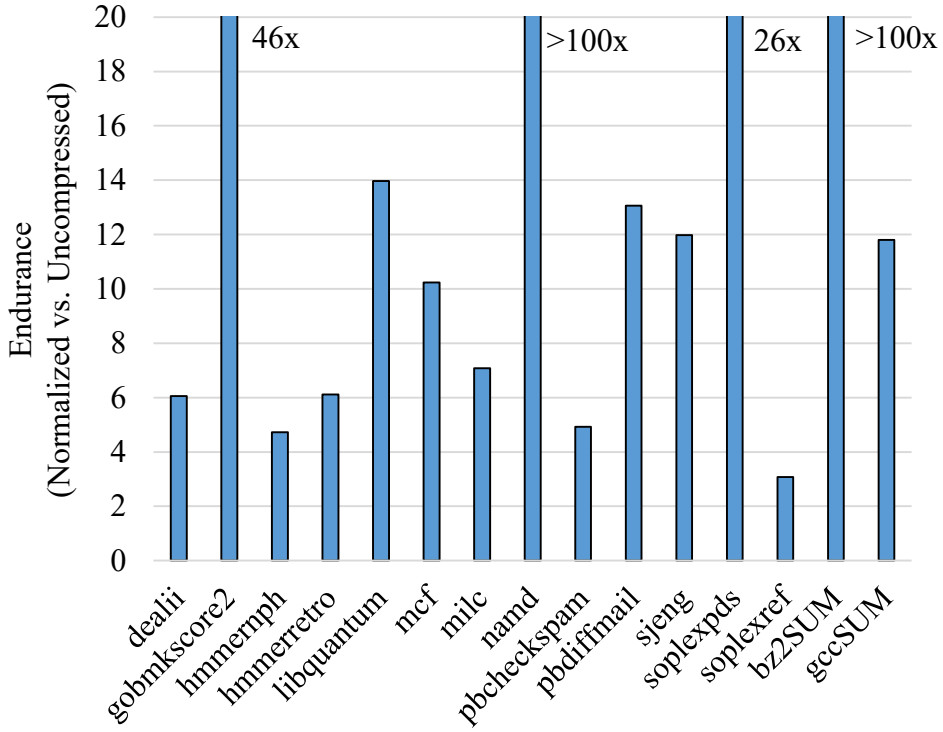


Figure 5.14: Endurance improvement of hybrid + Huffman + DCW cache versus uncompressed MRAM cache.

an average of somewhere around 20 times (2000%). Some benchmarks see an improvement in the high multiples of ten even.

Compared to the monolithic MRAM + DCW scheme, the improvement is significantly less, because the difference scheme does such a good job reducing writes. However, the Huffman + DCW scheme still shows improvement, gaining about 70% versus DCW. Figure 5.15 shows the results.

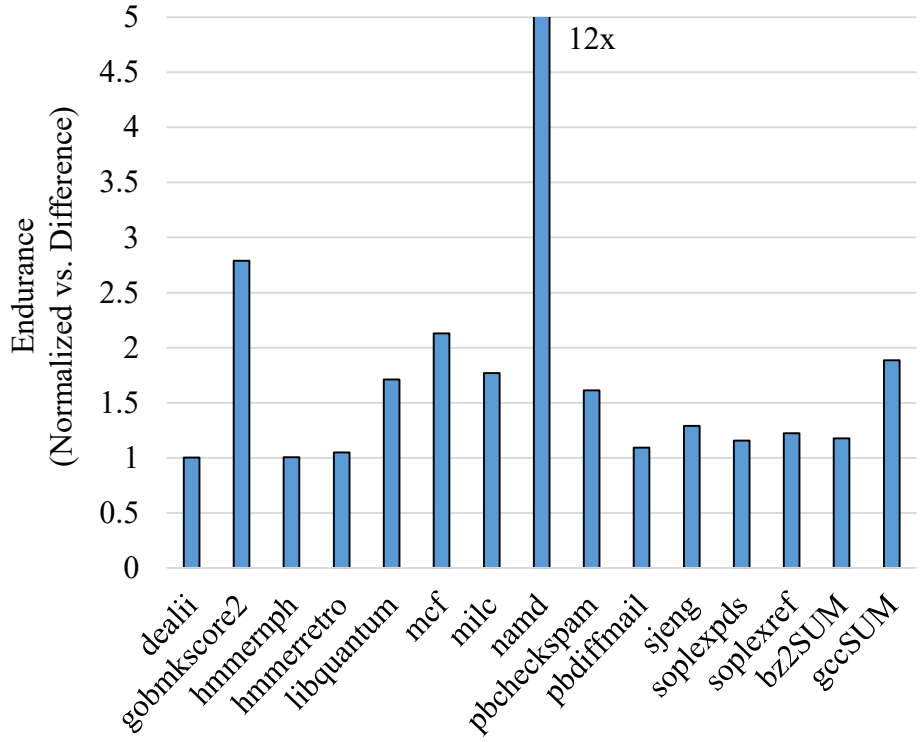


Figure 5.15: Endurance improvement of hybrid + Huffman + DCW cache versus MRAM + DCW cache.

5.5.4 Future Study

While we have shown that a hybrid cache configuration, with compression, can save significant amounts of energy over a monolithic cache, there are other interesting questions. For example, we can study different SRAM/MRAM fractions that make up the cache. One idea would be to size the SRAM smaller than the largest compressed value, causing additional writes to non-volatile memory, but saving more static energy.

Different memory technologies could also be investigated as they come to fruition. For example, as RRAM energy estimates become clearer, we can use them as well.

5.6 Related Work

5.6.1 Cache Energy Reduction

Caceh energy reduction is a favorite research topic [31]. A recent 2014 survey cites 150 papers on the subject. To reduce leakage energy, popular options are to put idle cache lines a "drowsy" (state-preserving) condition, or simply turned off (state-destroying). These solutions need extra circuitry and power tuning in order to function properly, adding complexity to the design. Our hybrid design avoids the circuitry needed to manage these power states, taking advantage of non-volatile memory's low static energy, and using a much smaller SRAM in order to minimize static energy for the cache as a whole.

Using compression to reduce static energy in an SRAM cache was proposed by K. Tanaka in 2006 and 2007 [46, 45]. It covers a monolithic cache architecture using compression for static energy savings, using various compression schemes. At the cache line level, if a cache line can be compressed to 3/4, 1/2, or 1/4 of its original size, the rest of the line is power gated. Our proposal considers finer granularities and exploits different memories in tandem, avoiding complex power gating schemes. Also, an uncompressed line using the proposed scheme would consume less static energy than in a monolithic architecture.

5.6.2 Cache Compression

Cache compression is a well studied topic. A recent survey cites 94 works [30]. The majority of these proposals compress multiple logical cache lines into the

physical space of one line. The proposals aim to increase the effective capacity of a cache, improving performance while keeping area and energy lower than a larger capacity cache. This work does not share the aim of increasing cache capacity. Instead, this proposal uses compression to turn off unneeded cache lines, reducing energy for a given capacity. This proposal avoids the complexity needed for caches that compress multiple lines into one (i.e. complex tag tracking and disassociating the tag and data arrays) as well.

5.7 Conclusion

In this part of the thesis, we describe a hybrid memory cache that exploits compression to reduce cache energy usage, taking advantage of SRAM and MRAM's complementary characteristics to convert costly non-volatile memory writes into cheaper SRAM writes. This is possible because of the tandem use of disparate memories. We replaced a monolithic SRAM or MRAM cache with a hybrid cache of equivalent capacity and modeled the energy savings. We found that a hybrid cache could save an average of 52% versus monolithic SRAM caches, and an additional 14% versus monolithic MRAM caches that use compression to reduce writes.

CHAPTER 6

CONCLUSION

Memory use and demand will continue to rise as we ask more and more of our computing platforms. To satisfy this demand while remaining within energy and area budgets is challenging, and architects have invented myriad ways to optimize and reduce the costs of adding more memory to computers today.

Upcoming non-volatile memories show a lot of promise, because they alter the trade-offs architects can make when designing memory systems. In particular, they offer lower static energy, at the cost of high dynamic write energy.

To exploit the advantages of both traditional memories (SRAM and DRAM) and the new non-volatile memories, architects have investigated hybrid memories, which are memory structures comprised of cells from disparate types of memory technologies.

In my thesis, I put forth a classification system for hybrid memories, giving a framework to the design space, intended to be useful for further research and discussion. I contribute three hybrid memory designs that show the promise and advantages of hybrid memories:

- Hybrid SRAM-Flash for non-volatile computing
- Hybrid SRAM-DRAM for multi-threaded register files
- Hybrid SRAM-MRAM for energy-efficient caches using compression

Each project shows significant energy and/or area savings versus a monolithic traditional memory array.

The SRAM-Flash design demonstrates a much more energy efficient method to provide non-volatile store and restore in energy-limited application domains. The two memories are integrated at a per-cell level, providing low-energy data movement. In the end, the proposed architecture enables extremely cheap and low latency non-volatile store and restore where there was no such capability before.

The second project, the per-cell hybrid memory comprised of SRAM and DRAMs, is able to replace very large SRAM register files with a smaller hybrid memory, while maintaining performance in the target application, fine-grained multi-threaded architectures.

Finally, a hybrid cache using SRAM and MRAM is able to reduce cache energy usage by attacking static energy (versus monolithic SRAM caches) and dynamic write energy (versus monolithic MRAM caches), by using compression to convert non-volatile writes into cheaper SRAM writes.

The promise of hybrid memories is here to stay as long as disparate memories have complementary traits to exploit. Until the rise of a truly universal memory, architects will be balancing and deciding memory hierarchies yet to come. Hybrid memories are a powerful design choice that enable computer architects to exploit the advantages of different memory types while mitigating or eliminating their disadvantages.

APPENDIX A

DETAILED ENERGY MODEL FOR HYBRID CACHE EVALUATION

In this Appendix we describe the energy model used for the hybrid cache with compression project, described in Chapter 5.

A.1 General Cache Energy Models

To analyze cache energy use, we rely on the straightforward model of static and dynamic energy:

$$Energy = Static\ Energy + Dynamic\ Energy$$

$$E_{total} = E_{static} + E_{dynamic}$$

A cache's static energy usage per unit time is mainly leakage, and is roughly proportional to area. Dynamic energy usage can be further split into read and write energy components.

$$Dynamic\ Energy = Read\ Energy + Write\ Energy$$

$$E_{dynamic} = E_{reads} + E_{writes}$$

For a monolithic, uncompressed, cache, we can use the read and write energies for that particular array size and technology, and multiply by the total amount of reads and writes, respectively. We assume an average activity value.

$$\begin{aligned} \textit{Dynamic Energy} = & \textit{Energy per Read} * \textit{No. of Reads} \\ & + \textit{Energy per Write} * \textit{No. of Writes} \end{aligned}$$

$$E_{dynamic} = E_{read} * N_{reads} + E_{write} * N_{writes}$$

A.2 Compressed Cache Energy Modeling

For a compressed cache, we have to consider a few more things. An uncompressed read, compressed read, uncompressed write, and compressed write, all have different costs. In addition to the costs of reading and writing the memory array, there are also overheads from the compression scheme itself.

Let us consider each case, generally.

Uncompressed read: Uncompressed reads can be treated as a normal read, as in a cache without compression.

Uncompressed write: Uncompressed writes can be treated as a normal write, as in a cache without compression.

Compressed read: A compressed read will require decompression before being sent out of the cache. The actual read operation may also have different costs compared to an uncompressed read (i.e. fewer bits read). The decompression operation can be expressed separately as an additional term.

Compressed write: A compressed write requires compression before being written to the cache. The actual write operation may also be different than

an uncompressed write (i.e. fewer bits written). The cost of compression can be expressed separately as an additional term.

To model a cache's energy use, each case requires tallying of which reads and writes are compressed.

The dynamic energy model for a compressed cache can be expressed:

$$\begin{aligned} Compressed\ Cache\ Dynamic\ Energy &= Uncompressed\ Read\ Energy \\ &+ Uncompressed\ Write\ Energy \\ &+ Compressed\ Read\ Energy \\ &+ Compressed\ Write\ Energy \end{aligned}$$

$$E_{CC,dynamic} = E_{UncompReads} + E_{UncompWrites} + E_{CompReads} + E_{CompWrites}$$

$$Uncompressed\ Read\ Energy = Energy\ per\ Read * No.\ of\ Uncompressed\ Reads$$

$$E_{Uncomp\ Reads} = E_{read} * N_{Uncomp\ Reads}$$

$$Uncompressed\ Write\ Energy = Energy\ per\ Write * No.\ of\ Uncompressed\ Writes$$

$$E_{Uncomp\ Writes} = E_{write} * N_{Uncomp\ Writes}$$

$$Compressed Read Energy = Energy per Compressed Read * No. of Compressed Reads$$

$$E_{Comp\ Reads} = E_{Comp\ Read} * N_{Comp\ Reads}$$

$$CompressedWriteEnergy = EnergyperCompressedWrite * No.ofCompressedWrites$$

$$E_{Comp\ Writes} = E_{Comp\ Write} * N_{Comp\ Writes}$$

$$\begin{aligned} Energy\ per\ Compressed\ Read\ &= Energy\ to\ read\ Compressed\ value \\ &+ Energy\ to\ decompress\ value \end{aligned}$$

$$E_{CompRead} = E_{CompMemoryRead} + E_{Decompress}$$

$$\begin{aligned} Energy\ per\ Compressed\ Write\ &= Energy\ to\ write\ Compressed\ value \\ &+ Energy\ to\ compress\ value \end{aligned}$$

$$E_{CompWrite} = E_{CompMemoryWrite} + E_{Compress}$$

A.3 Hybrid Cache Energy Modeling

The hybrid memory we are considering uses two separate memories to store one cache line. The bytes of the line are split; one part is stored using one type of memory; the other is stored using another. Each memory type has a different read and write cost. We can restructure the energy per read/write equations to take into account the energies of the disparate memories (memory A and B):

$$\textit{Energy per Hybrid Read} = \textit{Energy to read from Memory A} + \textit{Energy to read from Memory B}$$

$$E_{Hread} = E_{readA} + E_{readB}$$

$$\textit{Energy per Hybrid Write} = \textit{Energy to write to Memory A} + \textit{Energy to write to Memory B}$$

$$E_{Hwrite} = E_{writeA} + E_{writeB}$$

The total energy of every write and read can be obtained by multiplication by N_{reads} and N_{writes}.

A.4 Combining Compression and Hybrid Memories

In our proposed scheme, we add compression to a hybrid memory cache. We design the cache such that compressed values are stored in memory A, and uncompressed values are stored in both memories A and B. Modifying the compressed cache equations, we arrive at:

$$\begin{aligned}\textit{Hybrid Memory Compressed Cache Dynamic Energy} &= \textit{Uncompressed Read Energy} \\ &+ \textit{Uncompressed Write Energy} \\ &+ \textit{Compressed Read Energy} \\ &+ \textit{Compressed Write Energy}\end{aligned}$$

$$E_{HMCC,dynamic} = E_{HUncompReads} + E_{HUncompWrites} + E_{HCompReads} + E_{HCompWrites}$$

$$\begin{aligned} \textit{Uncompressed Read Energy} &= \textit{Energy per Hybrid Read} \\ &\quad * \textit{No. of Uncompressed Reads} \end{aligned}$$

$$E_{H\ Uncomp\ Reads} = E_{H\ read} * N_{Uncomp\ Reads}$$

$$\begin{aligned} \textit{Uncompressed Write Energy} &= \textit{Energy per Hybrid Write} \\ &\quad * \textit{No. of Uncompressed Writes} \end{aligned}$$

$$E_{H\ Uncomp\ Writes} = E_{H\ write} * N_{Uncomp\ Writes}$$

$$\begin{aligned} \textit{Compressed Read Energy} &= \textit{Energy per Hybrid Compressed Read} \\ &\quad * \textit{No. of Compressed Reads} \end{aligned}$$

$$E_{H\ Comp\ Reads} = E_{H\ Comp\ Read} * N_{Comp\ Reads}$$

$$\begin{aligned} \textit{Compressed Write Energy} &= \textit{Energy per Hybrid Compressed Write} \\ &\quad * \textit{No. of Compressed Writes} \end{aligned}$$

$$E_{H\ Comp\ Writes} = E_{H\ Comp\ Write} * N_{Comp\ Writes}$$

$$\begin{aligned} \textit{Energy per Hybrid Compressed Read} &= \textit{Energy to read Memory A} \\ &\quad + \textit{Energy to decompress value} \end{aligned}$$

$$E_{H\,Comp\,Read} = E_{read\,A} + E_{Decompress}$$

$$\begin{aligned} Energy\ per\ Hybrid\ Compressed\ Write\ &= Energy\ to\ write\ Memory\ A \\ &+ Energy\ to\ compress\ value \end{aligned}$$

$$E_{H\,Comp\,Write} = E_{write\,A} + E_{Compress}$$

Combining and flattening the 4 Uncompressed/Compressed Read/Write equations to primitives, we arrive at:

Uncompressed Read Energy:

$$E_{HUncompReads} = (E_{readA} + E_{readB}) * N_{UncompReads}$$

Uncompressed Write Energy:

$$E_{HUncompWrites} = (E_{writeA} + E_{writeB}) * N_{UncompWrites}$$

Compressed Read Energy:

$$E_{HCompReads} = (E_{readA} + E_{Decompress}) * N_{CompReads}$$

Compressed Write Energy:

$$E_{HCompWrites} = (E_{writeA} + E_{Compress}) * N_{CompWrites}$$

A.5 Difference Comparison Schemes

Another cache energy saving scheme is to only write the bits that have changed. When performing a write, this requires a read and comparison logic, and then

writing only the changed bits. The goal of such a scheme is to minimize the cost of a write; some memory technologies are quite costly to write. The overhead of a read and compare is less than writing every bit.

Per write, the energy can be calculated as:

$$E_{diffwrite} = E_{read} + E_{compare} + E_{changedwrite}$$

To express $E_{changedwrite}$ in a fashion we can tally statistics with, we use:

$$E_{changedwrite} = E_{write} * F$$

where F indicates the fraction of bits that change between writes.

To calculate the total energy for writes with differences, we have to do a sum of every write. This was expressed as a multiplication for the cases without difference detection, because every write operation should consume a similar amount of energy (the energy primitives are expressed on a per-write fashion as well). For the difference case, each write can consume a different amount of energy because the amount of bits changed can vary per write. This leads to the equation:

$$\begin{aligned} E_{diff,writes} &= \sum_{n=0}^{N_{writes}} (E_{read} + E_{compare} + E_{write} * F_n) \\ &= N_{writes} * (E_{read} + E_{compare}) + E_{write} * \sum_{n=0}^{N_{writes}} F_n \end{aligned}$$

Because F is a fraction, namely, (actual bits written)/(bits possible)per write, we can sum both numerator and denominator to arrive at an aggregate fraction for every write. For the number of writes we see, if we know the the aggregate fraction of bits actually written, then we can treat the sum as a multiply.

A.6 Hybrid Memory Caches with Difference Scheme

The natural cache to apply a difference scheme to is a hybrid memory cache that uses write-expensive memories (i.e. non-volatile memories). Read is unchanged. Write sees more terms added.

$$\begin{aligned} \textit{Energy per Hybrid Read} &= \textit{Energy to read from Memory A} \\ &+ \textit{Energy to read from Memory B} \end{aligned}$$

$$E_{Hread} = E_{readA} + E_{readB}$$

$$\begin{aligned} \textit{Energy per Hybrid Write} &= \textit{Energy to write to Memory A} \\ &+ \textit{Energy to write to Memory B} \end{aligned}$$

$$E_{Hdiffwrite} = E_{diffwriteA} + E_{diffwriteB}$$

$$\begin{aligned} E_{Hdiffwrite} = & E_{readA} + E_{compareA} + E_{writeA} * F_{changedwritesA} \\ & + E_{readB} + E_{compareB} + E_{writeB} * F_{changedwritesB} \end{aligned}$$

A.7 Compressed Cache with Difference Schemes

Applying this to a compressed cache gives us the following equations. The special case to note is the Compressed Write w/ Difference. The energy of a read, compare, and write is now a sum instead of a strict multiplication.

Note for some compressed caches, an uncompressed write and compressed write may have similar overhead for compression – an incoming value has to be checked to see if it is compressible.

$$\begin{aligned} Compressed\ Cache\ w/\ Difference\ Dynamic\ Energy\ &= Uncompressed\ Read\ Energy \\ &+ Uncompressed\ Write\ Energy \\ &+ Compressed\ Read\ Energy \\ &+ Compressed\ Write\ Energy \end{aligned}$$

$$\begin{aligned} E_{CC,diff,dynamic} &= E_{UncompReads} + E_{UncompdiffWrites} \\ &+ E_{CompReads} + E_{CompdiffWrites} \end{aligned}$$

$$Uncompressed\ Read\ Energy\ =\ Energy\ per\ Read\ *\ No.\ of\ Uncompressed\ Reads$$

$$E_{Uncomp\ Reads} = E_{read} * N_{Uncomp\ Reads}$$

$$Uncompressed\ Write\ Energy\ =\ Energy\ per\ Write\ *\ No.\ of\ Uncompressed\ Writes$$

$$E_{Uncomp\ diff\ Writes} = E_{diff\ write} * N_{Uncomp\ Writes}$$

$$\Longrightarrow\ E_{Uncomp\ Writes} = E_{read} + E_{compare} + E_{write} * F_{changed\ Uncomp\ writes} * N_{Uncomp\ Writes}$$

$$\begin{aligned} Compressed\ Read\ Energy\ &= Energy\ per\ Compressed\ Read \\ &* No.\ of\ Compressed\ Reads \end{aligned}$$

$$E_{Comp\ Reads} = E_{Comp\ Read} * N_{Comp\ Reads}$$

$$\begin{aligned} Compressed\ Write\ Energy\ &= Energy\ per\ Compressed\ Write \\ &* No.\ of\ Compressed\ Writes \end{aligned}$$

$$E_{Comp\ diff\ Writes} = E_{Comp\ diff\ Write} * N_{Comp\ Writes}$$

$$\begin{aligned} Energy\ per\ Compressed\ Read\ &= Energy\ to\ read\ Compressed\ value \\ &+ Energy\ to\ decompress\ value \end{aligned}$$

$$E_{Comp\ Read} = E_{read} + E_{Decompress}$$

$$\begin{aligned} Energy\ per\ Compressed\ Write\ &= Energy\ to\ write\ Compressed\ value \\ &+ Energy\ to\ compress\ value \end{aligned}$$

$$E_{Comp\ diff\ Write} = E_{Comp\ Memory\ diff\ Write} + E_{Compress}$$

$$\implies E_{Comp\ diff\ Write} = E_{read} + E_{compare} + E_{write} * F_{changed\ Comp\ writes} + E_{Compress}$$

A.8 Combining Hybrid Memories, Compression, and Difference Schema

Combining all three schema results in these equations:

$$\begin{aligned}\text{Hybrid Memory Compressed Cache Dynamic Energy} &= \text{Uncompressed Read Energy} \\ &+ \text{Uncompressed Write Energy} \\ &+ \text{Compressed Read Energy} \\ &+ \text{Compressed Write Energy}\end{aligned}$$

$$E_{HMCC,diff,dynamic} = E_{HUncompReads} + E_{HUncompdiffWrites} + E_{HCompReads} + E_{HCompdiffWrites}$$

$$\begin{aligned}\text{Uncompressed Read Energy} &= \text{Energy per Hybrid Read} \\ &* \text{No. of Uncompressed Reads}\end{aligned}$$

$$E_{HUncompReads} = E_{Hread} * N_{UncompReads}$$

$$\begin{aligned}\text{Uncompressed Write Energy} &= \text{Energy per Hybrid Write} \\ &* \text{No. of Uncompressed Writes}\end{aligned}$$

$$\begin{aligned}E_{HUncompdiffWrites} &= E_{Hdiffwrite} * N_{UncompWrites} \\ &= (E_{readA} + E_{compareA} + E_{writeA}) * F_{changeduncompwritesA} \\ &+ (E_{readB} + E_{compareB} + E_{writeB}) * F_{changeduncompwritesB} * N_{UncompWrites}\end{aligned}$$

$$\begin{aligned}\text{Compressed Read Energy} &= \text{Energy per Hybrid Compressed Read} \\ &* \text{No. of Compressed Reads}\end{aligned}$$

$$E_{HCompReads} = E_{HCompRead} * N_{CompReads}$$

$$\begin{aligned} Compressed\ Write\ Energy\ &= Energy\ per\ Hybrid\ Compressed\ Write \\ &* No.\ of\ Compressed\ Writes \end{aligned}$$

$$E_{HCompdiffWrites} = E_{HCompdiffWrite} * N_{CompWrites}$$

$$\begin{aligned} Energy\ per\ Hybrid\ Compressed\ Read\ &= Energy\ to\ read\ Memory\ A \\ &+ Energy\ to\ decompress\ value \end{aligned}$$

$$E_{HCompRead} = E_{readA} + E_{Decompress}$$

$$\begin{aligned} Energy\ per\ Hybrid\ Compressed\ Write\ &= Energy\ to\ write\ Memory\ A \\ &+ Energy\ to\ compress\ value \end{aligned}$$

$$\begin{aligned} E_{HCompdiffWrite} &= E_{diffwriteA} + E_{Compress} \\ &= E_{readA} + E_{compareA} + E_{writeA} * F_{changedcompwritesA} + E_{Compress} \end{aligned}$$

Combining and flattening the 4 Uncompressed/Compressed Read/Write equations to primitives, we arrive at:

Uncompressed Read Energy:

$$E_{HUncompReads} = (E_{readA} + E_{readB}) * N_{UncompReads}$$

Uncompressed Write Energy:

$$E_{HUncompdiffWrites} = (E_{readA} + E_{compareA} + E_{writeA} * F_{changeduncompwritesA} + E_{readB} + E_{compareB} + E_{writeB} * F_{changeduncompwritesB}) * N_{UncompWrites}$$

Compressed Read Energy:

$$E_{HCompReads} = (E_{readA} + E_{Decompress}) * N_{CompReads}$$

Compressed Write Energy:

$$E_{HCompdiffWrites} = (E_{readA} + E_{compareA} + E_{writeA} * F_{changedcompwritesA} + E_{Compress}) * N_{CompWrites}$$

BIBLIOGRAPHY

[1] Alaa R Alameldeen and David A Wood. Frequent pattern compression: A significance-based compression scheme for l2 caches. 2004.

[2] A. Arelakis and P. Stenstrom. Sc2: A statistical compression cache scheme. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 145–156, June 2014.

[3] Atmel. ATtiny4/5/9/10 preliminary summary, February 2010.

[4] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163 –174, April 2009.

[5] J. Barth, W. Reohr, P. Parries, G. Fredeman, J. Golz, S. Schuster, H. Matick, R. Hunter, C. Tanner J Harig, H. Kim, B. Khan, J. Griesemer, R. Havreluk, K. Yanagisawa, T. Kirihata, and S. Iyer. A 500MHz random cycle 1.5ns-latency, SOI embedded DRAM macro featuring a 3T micro sense amplifier. In *Proceedings of the IEEE International Symposium on Solid-State Circuits Conference (ISSCC)*, pages 486–617, Februrary 2007.

[6] R.A. Bheda, J.A. Poovey, J.G. Beu, and T.M. Conte. Energy efficient phase change memory based main memory for future high performance systems. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8, July 2011.

[7] Xiuyuan Bi, Mengjie Mao, Danghui Wang, and Hai Li. Unleashing the potential of mlc stt-ram caches. In *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, pages 429–436, Nov 2013.

[8] B.H. Calhoun, D.C. Daly, N. Verma, D.F. Finchelstein, D.D. Wentzloff, A. Wang, S-H. Cho, and A.P. Chandrakasan. Design considerations for ultra-low energy wireless microsensor nodes. *IEEE Transactions on Computers*, 54(6), June 2005.

[9] Mu-Tien Chang, P. Rosenfeld, Shih-Lien Lu, and B. Jacob. Technology comparison for large last-level caches (l3cs): Low-leakage sram, low write-energy stt-ram, and refresh-optimized edram. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 143–154, Feb 2013.

[10] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE Workload Characterization Symposium*, pages 44–54, 2009.

[11] Shuai Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, Liang Wang, and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 1–11, December 2010.

[12] Ki-Hwan Choi, Jong-Min Park, Jin-Ki Kim, Tae-Sung Jung, and Kang-Deog Suh. Floating-well charge pump circuits for sub-2.0v single power supply flash memories. In *VLSI Circuits, 1997. Digest of Technical Papers., 1997 Symposium on*, pages 61 –62, 12-14 1997.

[13] eetimes.com. Freescale: Thin-film flash mimics EEPROM, March 2010.

[14] H.R. Ghasemi, S.C. Draper, and Nam Sung Kim. Low-voltage on-chip cache architecture using heterogeneous cell sizes for high-performance processors. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 38–49, Feb 2011.

[15] Jeremy Gummeson, Shane S. Clark, Kevin Fu, and Deepak Ganesan. On the limits of effective hybrid micro-energy harvesting on mobile crfid sensors. In *Proceedings of the* 8^{th} *Annual International Conference on Mobile Systems, Applications and Services*, 2010.

[16] H. M. Haynie, J. M. Turner, J. C. Hanscom, M. Cadigan, N. Hadzic, D. Di Genova, J. Aylward, S. W. Salisbury, P. Sciuto, T. D. Needham, C. E. Bubb, and R. B. Tremaine. IBM system z10 open systems adapter ethernet data router. *IBM Journal of Research and Development*, 53(1):8:1 –8:12, January 2009.

[17] Jinwook Jung, Y. Nakata, M. Yoshimoto, and H. Kawaguchi. Energy-efficient spin-transfer torque ram cache exploiting additional all-zero-data flags. In *Quality Electronic Design (ISQED), 2013 14th International Symposium on*, pages 216–222, March 2013.

[18] David Kanter. Inside Fermi: Nvidia's HPC Push, 2009. http://www.realworldtech.com/ page.cfm?ArticleID=RWT093009110932.

[19] David Kanter. NVIDIA's GT200: Inside a Parallel Processor, 2009. http://www.realworldtech.com/ page.cfm?ArticleID=RWT090808195242.

[20] David Kanter. AMD's Cayman GPU architecture, 2010. http://realworldtech.com/ page.cfm?ArticleID=RWT121410213827.

[21] Pablo Bleyer Kocik. PacoBlaze - a synthesizable behavioral Verilog PicoBlaze clone, May 2007.

[22] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[23] Love Kothari and Nicholas P. Carter. Architecture of a self-checkpointing microprocessor that incorporates nanomagnetic devices. *IEEE Transactions on Computers*, 56(2), February 2007.

[24] S. Kvatinsky, Y.H. Nacson, Y. Etsion, E.G. Friedman, A. Kolodny, and U.C. Weiser. Memristor-based multithreading. *Computer Architecture Letters*, 13(1):41–44, Jan 2014.

[25] DongHyuk Lee, Yoongu Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu. Tiered-latency dram: A low latency and low cost dram architecture. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 615–626, Feb 2013.

[26] Zhi Li, Jingweijia Tan, and Xin Fu. Hybrid cmos-tfet based register files for energy-efficient gpgpus. In *Quality Electronic Design (ISQED), 2013 14th International Symposium on*, pages 112–119, March 2013.

[27] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[28] Kaisheng Ma, Yang Zheng, Shuangchen Li, K. Swaminathan, Xueqing Li, Yongpan Liu, J. Sampson, Yuan Xie, and V. Narayanan. Architecture exploration for ambient energy harvesting nonvolatile processors. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 526–537, Feb 2015.

[29] Richard E. Matick and Stanley E. Schuster. Logic-based eDRAM: Origins and rationale for use. *IBM Journal of Research and Development*, 49(1):145–165, January 2005.

[30] S. Mittal and J. Vetter. A survey of architectural approaches for data compression in cache and main memory systems. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1, 2015.

[31] Sparsh Mittal. A survey of architectural techniques for improving cache power efficiency. *Sustainable Computing: Informatics and Systems*, 4(1):33–43, 2014.

[32] T. Miwa, J. Yamada, H. Koike, H. Toyoshima, K. Amanuma, S. Kobayashi, T. Tatsumi, Y. Maejima, H. Hada, and T. Kunio. NV-SRAM: A nonvolatile SRAM with backup ferroelectric capacitors. *IEEE Journal of Solid-State Circuits*, 36(3), March 2001.

[33] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 3–14, Washington, DC, USA, 2007. IEEE Computer Society.

[34] Umesh Gajanan Nawathe, Mahmudul Hassan, King C. Yen, Ashok Kumar, Aparna Ramachandran, and David Greenhill. Implementation of an 8-core, 64-thread, power-efficient SPARC server on a chip. *IEEE Journal of Solid-State Circuits*, 43(1), January 2008.

[35] NVIDIA Corporation. NVIDIA CUDA Programming Guide, 2.0 edition, 2008.

[36] David A. Patterson and Carlo H. Sequin. RISC I: A reduced instruction set VLSI computer. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 443–457, 1981.

[37] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[38] Shantanu Rajwade, Wing kei Yu, Sarah Xu, Tuo-Hung Hou, G. Edward Suh, and Edwin Kan. Low power nonvolatile SRAM circuit with integrated

low voltage nanocrystal PMOS flash. In *Proceedings of the 23rd IEEE International System-On-Chip Conference*, 2010.

[39] N. Sakimura, T. Sugibayashi, R. Nebashi, and N. Kasai. Nonvolatile magnetic flip-flop for standby-power-free socs. *IEEE Journal of Solid-State Circuits*, 44(8), August 2009.

[40] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4):505–521, July 2005.

[41] Jon Stokes. Microsoft beats Intel, AMD to market with CPU/GPU combo chip, 2009. http://arstechnica.com/gaming/news/2010/08/microsoft-beats-intel-amd-to-market-with-cpugpu-combo-chip.ars.

[42] Guangyu Sun, Xiangyu Dong, Yuan Xie, Jian Li, and Yiran Chen. A novel architecture of the 3d stacked mram l2 cache for cmps. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 239–249, Feb 2009.

[43] Zhenyu Sun, Xiuyuan Bi, Hai (Helen) Li, Weng-Fai Wong, Zhong-Liang Ong, Xiaochun Zhu, and Wenqing Wu. Multi retention level stt-ram cache designs with a dynamic refresh scheme. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 329–338, New York, NY, USA, 2011. ACM.

[44] M. Takata, K. Nakayama, T. Izumi, T. Shinmura, J. Akita, and A. Kitagawa. Nonvolatile SRAM based on phase change. In *Proceedings of Nonvolatile Semiconductor Memory Workshop*, February 2006.

[45] K. Tanaka. Cache memory architecture for leakage energy reduction. In *Innovative architecture for future generation high-performance processors and systems, 2007. iwia 2007. international workshop on*, pages 73–80, Jan 2007.

[46] K. Tanaka and A. Matsuda. Static energy reduction in cache memories using data compression. In *TENCON 2006. 2006 IEEE Region 10 Conference*, pages 1–4, Nov 2006.

[47] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, January 2002.

[48] TSMC. Embedded memory, 2010. http://www.tsmc.com/download/brochures/2010_Embedded_Memory.pdf.

[49] UMC. Embedded memory, 2011. http://www.umc.com/english/pdf/eMemory.pdf.

[50] Alejandro Valero, Julio Sahuquillo, Salvador Petit, Vicente Lorente, Ramon Canal, Pedro López, and José Duato. An hybrid eDRAM/SRAM macrocell to implement first-level data caches. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 213–221, 2009.

[51] W. Wang, A. Gibby, Z. Wang, T. W. Chen, S. Fujita, P. Griffin, Y. Nishi, and S. Wong. Nonvolatile SRAM cell. In *IEDM Tech. Dig.*, December 2006.

[52] Yiqun Wang, Yongpan Liu, Shuangchen Li, Daming Zhang, Bo Zhao, Mei-Fang Chiang, Yanxin Yan, Baiko Sai, and Huazhong Yang. A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops. In *ESSCIRC (ESSCIRC), 2012 Proceedings of the*, pages 149–152, Sept 2012.

[53] Zhe Wang, D.A. Jimenez, Cong Xu, Guangyu Sun, and Yuan Xie. Adaptive placement and migration policy for an stt-ram-based hybrid cache. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 13–24, Feb 2014.

[54] D. Wendel, R. Kalla, R. Cargoni, J. Clables, J. Friedrich, R. Frech, J. Kahle, B. Sinharoy, W. Starke, S. Taylor, S. Weitzel, S.G. Chu, S. Islam, and V. Zyuban. The implementation of POWER7^{TM}: A highly parallel and scalable multi-core high-end server processor. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, pages 102 –103, February 2010.

[55] Xiaoxia Wu, Jian Li, Lixin Zhang, Evan Speight, Ram Rajamony, and Yuan Xie. Design exploration of hybrid caches with disparate memory technologies. *ACM Trans. Archit. Code Optim.*, 7(3):15:1–15:34, December 2010.

[56] Byung-Do Yang, Jae-Eun Lee, Jang-Su Kim, Junghyun Cho, Seung-Yun Lee, and Byoung gon Yu. A low power phase-change random access memory using a data-comparison write scheme. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 3014–3017, May 2007.

[57] Qing Yang and Jin Ren. I-cash: Intelligently coupled array of ssd and hdd. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 278–289, Feb 2011.